Rozumienie języka maszynowego

Dennis Yurichev

Rozumienie języka maszynowego

(Inżynieria wsteczna dla początkujących)

Dlaczego aż dwa tytuły? Przeczytaj tutaj: on page vi.

Dennis Yurichev
my emails



Wersja tekstu (23 października 2023).

Aktualna wersja (a także wersja rosyjska) tego tekstu znajduje się na https://beginners.re/.

Potrzebujemy tłumaczy!

Jeśli chcesz pomóc z tłumaczeniem tej książki na języki inne niż angielski i rosyjski, prześlij mailem fragment przetłumaczonego tekstu (obojętnie jakiej długości) a ja go dodam do kodu źródłowego.

Nie pytaj czy powinieneś tłumaczyć. Po prostu zrób coś. Przestałem odpowiadać na wiadomości “co powinienem zrobić”.

Czytać tutaj.

Statystyka tłumaczeń na inne języki znajduje się tutaj: https://beginners.re/.

Prędkość nie gra roli, jako że jest to projekt open-source. Twoje imię pojawi się obok imion innych uczestników projektu. Koreański, chiński i perski są zarezerwowane przez wydawców. Wersją angielską i rosyjską zajmuję się sam, ale mój angielski nadal jest daleki od ideału, więc będę wdzięczny za korekty. Nawet mój rosyjski nie jest idealny, więc będę wdzięczny również za korekty tekstu w języku rosyjskim!

Śmiało piszcie do mnie: my emails.

Skrócony spis treści

Spis treści

Przedmowa

Skąd dwa tytuły?

W latach 2014-2018 książka nosiła tytuł "Inżynieria wsteczna dla początkujących" ale zawsze podejrzewałem, że zawęża to grono czytelników.

Ludzie zajmujący się bezpieczeństwem informacji (infosec) wiedzą o inżynierii wstecznej, jednak rzadko kiedy słyszałem, by używali słowa asembler.

Podobnie, termin "inżynieria wsteczna" jest nieco tajemniczy dla ogólnego grona programistów, jednak wiedzą oni o istnieniu asemblera.

W lipcu 2018 roku, w ramach eksperymentu, zmieniłem tytuł na "Język maszynowy dla początkujących" i umieściłem link na portalu Hacker News[1], i książka została ogólnie dobrze przyjęta.

Niech więc tak będzie, książka ma teraz dwa tytuły.

Zmieniłem jednak drugi tytuł na "Rozumienie kodu maszynowego", ponieważ ktoś już napisał książkę o tytule "Język maszynowy dla początkujących". Ludzie twierdzą, że "dla początkujących" brzmi odrobinę ironicznie jak na ~1000 stronicową książkę.

Dwie książki różnią się jedynie tytułem, nazwą pliku (UAL-XX.pdf versus RE4B-XX.pdf), URLem i kilkoma pierwszymi stronami.

O inżynierii wstecznej

Termin „inżynieria wsteczna" ma kilka popularnych definicji:

1) inżynieria wsteczna oprogramowania; analiza skompilowanych programów;

2) skanowanie modelu w 3D, żeby następnie go skopiować;

3) odzyskiwanie struktury DBMS[2].

Nasza książka będzie powiązana z tą pierwszą definicją.

Pożądana wiedza

Podstawowa znajomość C PL[3] . Polecane materiały: 7.1.3 on page 150.

Ćwiczenia i zadania

…wszystkie są na osobnej stronie: http://challenges.re.

Pochwały dla książki

https://beginners.re/#praise.

[1] https://news.ycombinator.com/item?id=17549050
[2] Database Management Systems
[3] Język programowania (Programming Language)

Uczelnie

Ta książka jest polecana przynajmniej na poniższych uczelniach: https://beginners.re/#uni.

Podziękowania

Za cierpliwe odpowiadanie na wszystkie moje pytania: SkullC0DEr.

Za wskazanie błędów i nieścisłości: Alexander Lysenko, Federico Ramondino, Mark Wilson, Razikhova Meiramgul Kayratovna, Anatoly Prokofiev, Kostya Begunets, Valentin "netch" Nechayev, Aleksandr Plakhov, Artem Metla, Alexander Yastrebov, Vlad Golovkin[4], Evgeny Proshin, Alexander Myasnikov, Alexey Tretiakov, Oleg Peskov, Pavel Shakhov, Zhu Ruijin, Changmin Heo, Vitor Vidal, Stijn Crevits, Jean-Gregoire Foulon[5], Ben L., Etienne Khan, Norbert Szetei[6], Marc Remy, Michael Hansen, Derk Barten, The Renaissance[7], Hugo Chan, Emil Mursalimov, Tanner Hoke, Tan90909090@GitHub, Ole Petter Orhagen, Sourav Punoriyar, Vitor Oliveira, Alexis Ehret, Maxim Shlochiski, Greg Paton, Pierrick Lebourgeois, Abdullah Alomair, Bobby Battista, Ashod Nakashian.

Za inną pomoc: Andrew Zubinski, Arnaud Patard (rtp on #debian-arm IRC), noshadow on #gcc IRC, Aliaksandr Autayeu, Mohsen Mostafa Jokar, Peter Sovietov, Misha "tiphareth" Verbitsky.

Za przetłumaczenie tej książki na język chiński uproszczony: Antiy Labs (antiy.cn), Archer.

Za tłumaczenie na język koreański: Byungho Min.

Za tłumaczenie na język holenderski: Cedric Sambre (AKA Midas).

Za tłumaczenie na język hiszpański: Diego Boy, Luis Alberto Espinosa Calvo, Fernando Guida, Diogo Mussi, Patricio Galdames, Emiliano Estevarena.

Za tłumaczenie na język portugalski: Thales Stevan de A. Gois, Diogo Mussi, Luiz Filipe, Primo David Santini.

Za tłumaczenie na język włoski: Federico Ramondino[8], Paolo Stivanin[9], twyK, Fabrizio Bertone, Matteo Sticco, Marco Negro[10], bluepulsar.

Za tłumaczenie na język francuski: Florent Besnard[11], Marc Remy[12], Baudouin Landais, Téo Dacquet[13], BlueSkeye@GitHub[14].

---

[4] goto-vlad@github
[5] https://github.com/pixjuan
[6] https://github.com/73696e65
[7] https://github.com/TheRenaissance
[8] https://github.com/pinkrab
[9] https://github.com/paolostivanin
[10] https://github.com/Internaut401
[11] https://github.com/besnardf
[12] https://github.com/mremy
[13] https://github.com/T30rix
[14] https://github.com/BlueSkeye

Za tłumaczenie na język niemiecki: Dennis Siekmeier[15], Julius Angres[16], Dirk Loser[17], Clemens Tamme, Philipp Schweinzer, Tobias Deiminger.

Za tłumaczenie na język polski: Kateryna Rozanova, Aleksander Mistewicz, Wiktoria Lewicka, Marcin Sokołowski.

Za tłumaczenie na język japoński: shmz@github[18],4ryuJP@github[19].

Za korektę: Vladimir Botov, Andrei Brazhuk, Mark "Logxen" Cooper, Yuan Jochen Kang, Mal Malakov, Lewis Porter, Jarle Thorsen, Hong Xie.

Vasil Kolev[20] wprowadził wiele poprawek i wskazał sporo błędów.

Dziękuję również wszystkim użytkownikom z github.com za ich komentarze i poprawki.

Użyłem wielu pakietów LaTeX. Chciałbym podziękować również ich autorom.

Darczyńcy

Tym wszystkim, którzy mnie wspierali w czasie pisania tej książki:

2 * Oleg Vygovsky (50+100 UAH), Daniel Bilar ($50), James Truscott ($4.5), Luis Rocha ($63), Joris van de Vis ($127), Richard S Shultz ($20), Jang Minchang ($20), Shade Atlas (5 AUD), Yao Xiao ($10), Pawel Szczur (40 CHF), Justin Simms ($20), Shawn the R0ck ($27), Ki Chan Ahn ($50), Triop AB (100 SEK), Ange Albertini (€10+50), Sergey Lukianov (300 RUR), Ludvig Gislason (200 SEK), Gérard Labadie (€40), Sergey Volchkov (10 AUD), Vankayala Vigneswararao ($50), Philippe Teuwen ($4), Martin Haeberli ($10), Victor Cazacov (€5), Tobias Sturzenegger (10 CHF), Sonny Thai ($15), Bayna AlZaabi ($75), Redfive B.V. (€25), Joona Oskari Heikkilä (€5), Marshall Bishop ($50), Nicolas Werner (€12), Jeremy Brown ($100), Alexandre Borges ($25), Vladimir Dikovski (€50), Jiarui Hong (100.00 SEK), Jim Di (500 RUR), Tan Vincent ($30), Sri Harsha Kandrakota (10 AUD), Pillay Harish (10 SGD), Timur Valiev (230 RUR), Carlos Garcia Prado (€10), Salikov Alexander (500 RUR), Oliver Whitehouse (30 GBP), Katy Moe ($14), Maxim Dyakonov ($3), Sebastian Aguilera (€20), Hans-Martin Münch (€15), Jarle Thorsen (100 NOK), Vitaly Osipov ($100), Yuri Romanov (1000 RUR), Aliaksandr Autayeu (€10), Tudor Azoitei ($40), Z0vsky (€10), Yu Dai ($10), Anonymous ($15), Vladislav Chelnokov ($25), Nenad Noveljic ($50), Ryan Smith ($25), Andreas Schommer (€5), Nikolay Gavrilov ($300), Ernesto Bonev Reynoso ($30).

bardzo dziękuję.

mini-FAQ

Q: Czy ta książka jest prostsza niż inne?

A: Nie, poziom trudności jest mniej więcej taki sam jak innych książek na ten temat.

[15] https://github.com/DSiekmeier
[16] https://github.com/JAngres
[17] https://github.com/PolymathMonkey
[18] https://github.com/shmz
[19] https://github.com/4ryuJP
[20] https://vasil.ludost.net/

Q: Obawiam się zacząć czytać tę książkę, ma ponad 1000 stron. ”... dla początkujących” w nazwie brzmi nieco ironicznie.

A: Wszelkiego rodzaju kody źródłowe stanowią większość tej książki. Ta książka naprawdę jest dla początkujących, wiele w niej (jeszcze) brakuje.

Q: Co trzeba wiedzieć zanim się przystąpi do czytania książki?

A: Umiejętności C/C++ są pożądane, ale nie są niezbędne.

Q: Czy powinienem uczyć się jednocześnie x86/x64/ARM i MIPS? Czy to nie za dużo?

A: Myślę, że na początek wystarczy czytać tylko o x86/x64, części o ARM i MIPS można pominąć.

Q: Czy można zakupić książki w wersji papierowej w języku rosyjskim lub angielskim?

A: Niestety nie, żaden wydawca jeszcze się nie zainteresował wydaniem rosyjskiej lub angielskiej wersji. Natomiast można ją wydrukować i zbindować w każdym ksero. https://yurichev.com/news/20200222_printed_RE4B/.

Q: Czy istnieje wersja epub/mobi?

A: Nie. W wielu miejscach książka korzysta z hacków specyficznych dla TeXa/LaTeXa, dlatego przerobienie jej na HTML (epub/mobi to jest HTML) nie jest łatwe.

Q: Po co uczyć się asemblera w dzisiejszych czasach?

A: Jeśli nie jest się programistą OS[21], to prawdopodobnie nie trzeba nic pisać w asemblerze: współczesne kompilatory optymalizują kod lepiej niż człowiek [22].

Do tego współczesne CPU[23] są bardzo skomplikowanymi urządzeniami i znajomość asemblera nie pomoże poznać ich mechanizmów wewnętrznych.

Jednak zostają dwa obszary, w których dobra znajomość asemblera może być pomocna: 1) badanie malware (złośliwego oprogramowania) w celu jego analizy ; 2) lepsze zrozumienie skompilowanego kodu w trakcie debuggowania.

Wobec tego ta książka jest napisana dla tych ludzi, którzy raczej chcą rozumieć assembler, a nie w nim pisać. Stąd jest w niej bardzo dużo przykładów - wyjść kompilatora.

Q: Kliknąłem w odnośnik wewnątrz pliku PDF, jak teraz wrócić?

A: W Adobe Acrobat Reader trzeba wcisnąć Alt+LeftArrow. W Evince wcisnąć ”<”.

Q: Czy mogę wydrukować tę książkę? Korzystać z niej do nauczania?

A: Oczywiście, właśnie dlatego ta książka ma licencję Creative Commons (CC BY-SA 4.0).

Q: Dlaczego ta książka jest darmowa? Wykonałeś świetną robotę. To podejrzane, podobnie jak z innymi rzeczami za darmo.

[21] System operacyjny (Operating System)

[22] Bardzo ciekawy artykuł na ten temat: [Agner Fog, *The microarchitecture of Intel, AMD and VIA CPUs*, (2016)]

[23] Central Processing Unit

A: Moim zdaniem autorzy literatury technicznej robią to dla autoreklamy. Taka praca nie przynosi za dużo pieniędzy.

Q: Jak znaleźć pracę w zawodzie reverse engineeraa?

A: Na reddit (RE[24]), od czasu od czasu pojawiają się wątki poszukiwania pracowników. Możesz spróbować tam poszukać.

Q: Mam pytanie...

A: Napisz do mnie maila (my emails).

O tłumaczeniu na język koreański

W styczniu 2015, wydawnictwo Acorn (www.acornpub.co.kr) z Korei Południowej wykonało ciężką pracę, żeby przetłumaczyć i wydać moją książkę (stanem na sierpień 2014) w języku koreańskim. Jest ona teraz dostępna na ich stronie.

Tłumaczył Byungho Min (twitter/tais9). Okładka namalował mój dobry przyjaciel, artysta, Andy Nechaevsky facebook/andydinka.

Acorn również ma prawa autorskie do tłumaczenia koreańskiego. Jakbyście chcieli mieć *prawdziwą* książkę w języku koreańskim i chcielibyście wesprzeć moją pracę, możecie ją kupić.

O tłumaczeniu na język perski (farsi)

W roku 2016 książkę przetłumaczył Mohsen Mostafa Jokar (znany w irańskiej społeczności z tłumaczenia instrukcji do Radare[25]) Książka jest dostępna na stronie wydawnictwa [26] (Pendare Pars).

Pierwsze 40 stron: `https://beginners.re/farsi.pdf`.

Pozycja książki w Narodowej Bibliotece Iranu: `http://opac.nlai.ir/opac-prod/bibliographic/4473995`.

O tłumaczeniu na język chiński

W kwietniu 2017, wydawnictwo PTPress skończyło tłumaczenie mojej książki na język chiński. Mają również prawo autorskie do tłumaczenia chińskiego.

Chińskie tłumaczenie można zamówić tutaj: `http://www.epubit.com.cn/book/details/4174`. Recenzje i historię tłumaczenia można znaleźć tutaj: `http://www.cptoday.cn/news/detail/3155`.

Głównym tłumaczem był Archer, u którego mam teraz dług wdzięczności. Był bardzo dociekliwy i znalazł w książce sporo bugów i błędów, co jest szczególnie ważne w literaturze, której dotyczy ta książka.

Będę polecał go również innym autorom!

[24] reddit.com/r/ReverseEngineering/
[25] `http://rada.re/get/radare2book-persian.pdf`
[26] `http://goo.gl/2Tzx0H`

Chłopaki z Antiy Labs również pomogli z tłumaczeniem. Tutaj słowo wstępne napisane przez nich.

Rozdział 1

Przykłady kodu

1.1 Metoda

Kiedy autor tej książki uczył się C, a później C++, pisał niewielkie kawałki kodu, kompilował i patrzył jak wyglądają w asemblerze. Tak było o wiele łatwiej zrozumieć co się dzieje w programie.[1]. Robił to wystarczająco dużo razy, żeby związek między kodem w C/C++ a tym co generuje kompilator wbił się w jego podświadomość bardzo głęboko. Dzięki temu łatwo mu określić zgrubną strukturę kodu w C, patrząc na kod asemblera. Możliwe, że ta metoda pomoże komuś jeszcze.

Czasami wykorzystam stare kompilatory, żeby otrzymać bardzo krótki lub prosty kawałek kodu.

Przy okazji, jest świetna strona, gdzie możesz zrobić to samo, używając różnych kompilatorów - bez konieczności instalowania ich u siebie: `http://godbolt.org/`.

Ćwiczenia

Kiedy autor tej książki uczył się asemblera, często kompilował krótkie funkcje w C i przepisywał je stopniowo na asemblera, starając się uzyskać jak najkrótszy kodu. Prawdopodobnie nie warto tego robić w praktycznych zastosowaniach, ponieważ trudno jest konkurować ze współczesnymi kompilatorami pod względem wydajności. Jest to jednak bardzo dobry sposób na zrozumienie asemblera. Możesz wziąć dowolny fragment kodu w asemblerze z tej książki i postarać się uczynić go krótszym. Ale nie zapomnij przetestować swojego rezultatu.

Poziomy optymalizacji i debuggowanie

Kod źródłowy można kompilować różnymi kompilatorami z różnym poziomami optymalizacji. W typowym kompilatorze jest tych poziomów około trzech, gdzie poziom

[1] Szczerze mówiąc, dalej tak robi, kiedy nie rozumie jak jakiś kod działa. Ostatni przykład, z 2019 roku: `p += p+(i&1)+2;` z “SAT0W”, SAT-solvera autorstwa D. Knutha.

zerowy oznacza wyłączoną optymalizację. Optymalizować można rozmiar programu lub jego szybkość. Kompilator, który nie dokonuje optymalizacji, działa szybciej i generuje bardziej przejrzysty kod (choć i większy objętościowo). Kompilator, który dokonuje optymalizacji, działa wolniej i stara się wygenerować jak najszybszy kod (co nie zawsze znaczy, że kod będzie krótszy). Obok poziomów i kierunków optymalizacji kompilator może załączać do pliku wynikowego dodatkowe informacje dla debuggera, tworząc w ten sposób kod, który jest prostszy w debuggowaniu. Bardzo ważną cechą kodu debuggowanego jest to, że może on zawierać powiązanie między każdą linią kodu źródłowego a adresem w kodzie maszynowym. Kompilatory, dokonując optymalizacji, zwykle generują kod, gdzie całe linie kodu źródłowego mogą zostać pominięte nie będą nawet widoczne w kodzie maszynowym. Praktykujący reverse engineer z reguły ma styczność z obiema wersjami, jako że niektórzy developerzy włączają optymalizację, a niektórzy - nie.

Dlatego będziemy pracować z przykładami kodu w obu wariantach.

1.2 Niektóre podstawowe pojęcia

1.2.1 Krótkie wprowadzenie do CPU

CPU (procesor) jest urządzeniem, które wykonuje bezpośrednio kod maszynowy programu.

Terminologia:

Instrukcja : prymitywny rozkaz CPU. Najprostsze przykłady: przenoszenie (kopiowanie) danych między rejestrami, korzystanie z pamięci (zapis/odczyt), proste operacje arytmetyczne.

Z reguły każdy CPU ma swój zestaw instrukcji (ISA[2]).

Kod maszynowy : kod wykonywany bezpośrednio przez CPU. Każda instrukcja kodu maszynowego zwykle jest kodowana za pomocą kilku bajtów.

Język asemblera : kod maszynowy plus niektóre rozszerzenia (np. makra), stworzone po to, żeby ułatwić pracę programiście.

Rejestr CPU : Każdy CPU ma swój zestaw rejestrów ogólnego przeznaczenia (GPR[3]). ≈ 8 w x86, ≈ 16 w x86-64 i ≈ 16 w ARM. Najłatwiej myśleć o rejestrze jak o zmiennej tymczasowej bez określonego typu. Wyobraź sobie, że pisząc w języku wyższego poziomu, masz dostępnych tylko 8 zmiennych o szerokości 32 (lub 64) bitów. Te 8 zmiennych to właśnie rejestry. Wbrew pozorom można z nimi naprawdę wiele zrobić!

Dlaczego występują języki niższego i wyższego poziomu? Odpowiedź jest prosta: ludzie i procesory różnią się między sobą - dla człowieka jest o wiele łatwiej pisać w wysokopoziomowym języku programowania typu C/C++, Java czy Python, a dla procesora łatwiej jest pracować na niższym poziomie abstrakcji. Zapewne można by zbudować procesor, który wykonywałby kod wysokiego poziomu, ale jego budowa byłaby dużo bardziej skomplikowana niż budowa procesorów jakie obecnie znamy. I

[2] Instruction Set Architecture (architektura listy rozkazów)
[3] General Purpose Registers (rejestry ogólnego przeznaczania)

odwrotnie, pisanie w języku asemblera jest dla ludzi bardzo niewygodne, z uwagi na jego niski poziom i trudność kodowania bez popełniania całej masy drobnych błędów. Program, który potrafi konwertować język wysokiego poziomu na kod asemblera, nazywamy *kompilatorem*.

Kilka słów o różnicy między ISA

x86 od zawsze zawierało instrukcje o różnej długości, więc kiedy nadeszła era 64-bitowej architektury, rozszerzenia x64 nie wpłynęły znacząco na ISA.

ARM to procesor RISC[4] zaprojektowany tak, żeby zawierał wszystkie instrukcje tej samej długości, co miało sporo zalet w przeszłości. Na samym początku wszystkie instrukcje ARM były kodowane na czterech bajtach (obecnie „tryb ARM")[5].

Później się okazało, że nie jest to zbyt ekonomiczne, bo najczęściej używane przez procesor instrukcje[6] mogą być zakodowane z wykorzystaniem mniejszej ilości informacji. Więc dodano inną ISA o nazwie "Thumb", gdzie każda instrukcja jest kodowana za pomocą tylko 2 bajtów. Jednak nie *wszystkie* instrukcje ARM mogą być zakodowane na 2 bajtach, więc zestaw instrukcji Thumb jest ograniczony. Kod skompilowany dla trybu ARM i Thumb może współdziałać w jednym programie.

Później twórcy ARM stwierdzili, że Thumb można rozszerzyć: tak pojawił się Thumb-2 (w ARMv7). Thumb-2 to wciąż dwubajtowe instrukcje, ale niektóre nowe instrukcje mają długość 4 bajtów. Szeroko rozpowszechnioną i błędną opinią jest to, że Thumb-2 to mieszanina ARM i Thumb. Tryb Thumb-2 został rozszerzony w celu wsparcia dla wszystkich możliwości procesora, by mógł konkurować z trybem ARM—co zostało w pełni osiągnięte, gdyż większość aplikacji na urządzenia iPod/iPhone/iPad są skompilowane dla zestawu instrukcji Thumb-2, (trzeba przyznać, że w dużej mierze jest to zasługa Xcode, który robi to domyślnie).

Później pojawił się 64-bitowy ARM. Jest to ISA znowu z 4-bajtowymi instrukcjami, bez dodatkowego trybu Thumb. Jednak nowe 64-bitowe wymagania wpłynęły na ISA tak, że obecnie mamy 3 zestawy instrukcji ARM: tryb ARM, tryb Thumb/Thumb-2 i ARM64. Te zestawy instrukcji częściowo się pokrywają, ale można powiedzieć, że są to różne zestawy a nie wariacje tego samego ISA. W tej książce postaramy się zaprezentować fragmenty kodu we wszystkich trzech trybach ISA. Istnieje jeszcze wiele innych RISC ISA z instrukcjami 32-bitowej długości — np. MIPS, PowerPC i Alpha AXP.

[4]Reduced Instruction Set Computing

[5]Instrukcje o stałym rozmiarze są wygodne, bo dzięki temu można łatwo znaleźć adres następnej (lub poprzedniej) instrukcji. Dokładniejsze wyjaśnienie znajduje się w sekcji o operatorze switch() (**??** on page ??).

[6]Są nimi MOV/PUSH/CALL/Jcc

1.2.2 Systemy liczbowe

> Nowadays octal numbers seem to be used for exactly one purpose—file permissions on POSIX systems—but hexadecimal numbers are widely used to emphasize the bit pattern of a number over its numeric value.
>
> Alan A. A. Donovan, Brian W. Kernighan — The Go Programming Language

Ludzie przyzwyczaili się do systemu dziesiętnego prawdopodobnie dlatego, że każdy ma 10 palców. Natomiast liczba 10 nie odgrywa szczególnej roli w nauce i matematyce. Binarny (dwójkowy) system liczbowy jest naturalny dla techniki cyfrowej i elektroniki: 0 oznacza brak prądu, a 1 — jego obecność. 10 w systemie binarnym to 2 w dziesiętnym; 100 w binarnym to 4 w dziesiętnym, itd.

Jeżeli w systemie liczbowym jest 10 znaków, jego *podstawa* (ang. *radix* lub *base*) to 10. System dwójkowy ma *podstawę* 2.

Ważne rzeczy, które warto sobie przypomnieć:

1) *liczba* jest liczbą, natomiast *cyfra* to umowny znak pisarski służący do zapisywania liczb; 2) sama w sobie liczba się nie zmienia przy przeliczaniu z jednego systemu na inny: zmienia się tylko sposób jej zapisu (lub reprezentacja w pamięci).

Jak skonwertować liczbę z jednego systemu na drugi?

Z notacji pozycyjnej korzysta się prawie wszędzie, to znaczy, że każda cyfra posiada swoją wagę w zależności od jej usytuowania wewnątrz liczby. Jeżeli 2 znajduje sie na ostatnim miejscu od prawej, jest to 2. Jeżeli jest ona usytuowana w miejscu przedostatnim, jest to 20.

Co oznacza zapis 1234?

$10^3 \cdot 1 + 10^2 \cdot 2 + 10^1 \cdot 3 + 1 \cdot 4$ = 1234 lub $1000 \cdot 1 + 100 \cdot 2 + 10 \cdot 3 + 4 = 1234$

Tak samo to wygląda w przypadku liczb binarnych, tyle że podstawą jest 2, a nie 10. Co oznacza zapis 0b101011?

$2^5 \cdot 1 + 2^4 \cdot 0 + 2^3 \cdot 1 + 2^2 \cdot 0 + 2^1 \cdot 1 + 2^0 \cdot 1 = 43$ lub $32 \cdot 1 + 16 \cdot 0 + 8 \cdot 1 + 4 \cdot 0 + 2 \cdot 1 + 1 = 43$

Notację pozycyjną można przeciwstawić notacji niepozycyjnej, np. rzymskiej. [7]. Prawdopodobnie ludzkość przeszła na notację pozycyjną, ponieważ w ten sposób łatwiej jest wykonywać proste operacje (dodawanie, mnożenie, itd.) na papierze, ręcznie.

Liczby binarne również można pisemnie dodawać, odejmować itd., dokładnie tak samo jak uczy się tego w szkole, tylko z użyciem dwóch cyfr.

Liczby w zapisie binarnym są nieporęczne, kiedy stosuje się je w kodzie źródłowym i zrzutach pamięci, dlatego w tych miejscach używa się systemu szesnastkowego (heksadecymalnego), o podstawie 16. System szesnastkowy używa szesnastu cyfr - znaków 0-9 oraz A-F. Każda cyfra zajmuje 4 bity lub 4 cyfry w systemie binarnym,

[7] O ewolucji systemów liczbowych przeczytasz w [Donald E. Knuth, *The Art of Computer Programming*, Volume 2, 3rd ed., (1997), 195–213.]

więc łatwo można konwertować z reprezentacji binarnej na szesnastkową i odwrotnie, nawet w pamięci.

szesnastkowy	binarny	dziesiętny
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15

Jak rozpoznać w jakim systemie zapisana jest konkretna liczba?

Liczby dziesiętne z reguły są zapisywane tradycyjnie, czyli 1234. Ale niektóre asemblery pozwalają podkreślić to przez sufiks "d": 1234d.

Do liczb binarnych czasami dodaje się prefiks "0b": 0b100110111 (W GCC[8] istnieje do tego niestandardowe rozszerzenie [9]). Jest jeszcze inny sposób: sufiks "b", np: 100110111b. W tej książce będę się trzymał konwencji prefiksowej "0b" dla liczb binarnych.

Liczby szesnastkowe mają prefiks "0x" w C/C++ i niektórych innych PL: 0x1234ABCD. Lub mają sufiks "h": 1234ABCDh — zwykle są w ten sposób reprezentowane w asemblerach lub debuggerach. W tej konwencji, jeśli liczba zaczyna się od A..F, przed nimi dopisuje się 0: 0ABCDEFh. Za czasów 8-bitowych komputerów domowych był również sposób zapisu liczb za pomocą prefiksu $, np., $ABCD. W tej książce będę się trzymał prefiksu "0x" dla liczb szesnastkowych.

Czy trzeba umieć konwertować liczby w głowie? Tablicę liczb szesnastkowych składających się z jednej cyfry łatwo zapamiętać, ale raczej nie warto zapamiętywać większych liczb.

Prawdopodobnie najczęściej liczby szesnastkowe są spotykane w URL[10]-ach. W ten sposób są kodowane litery spoza alfabetu łacińskiego. Np.: `https://en.wiktionary.org/wiki/na%C3%AFvet%C3%A9` to URL strony w Wiktionary o słowie „naïveté".

[8] GNU Compiler Collection
[9] `https://gcc.gnu.org/onlinedocs/gcc/Binary-constants.html`
[10] Uniform Resource Locator

System ósemkowy

Jeszcze jeden system, z którego często korzystało się w informatyce w przeszłości to system oktalny. System oktalny ma 8 cyfr (0-7), każda opisująca 3 bity, więc łatwo przeliczać liczbę na inne systemu, w obie strony. Ten system prawie wszędzie został zastąpiony przez szesnastkowy, ale, o dziwo, w systemach *NIX nadal jest program korzystający z systemu ósemkowego: `chmod`.

Jak wiedzą użytkownicy systemów *NIX, argumentem `chmod` jest liczba składająca się z 3 cyfr. Pierwsza cyfra określa uprawnienia właściciela pliku, druga - to uprawnienia grupy (do której plik należy), trzecia dla reszty użytkowników. Każda cyfra może być przedstawiona binarnie:

dziesiętny	binarny	znaczenie
7	111	**rwx**
6	110	**rw-**
5	101	**r-x**
4	100	**r--**
3	011	**-wx**
2	010	**-w-**
1	001	**--x**
0	000	**---**

Więc każdy bit jest powiązany z flagą: read/write/execute (prawo do odczytu/zapisu-/wykonania).

Właśnie dlatego wspomniałem o `chmod` — liczba, będąca argumentem, może być reprezentowana w systemie ósemkowym. Na przykład weźmy 644. Kiedy uruchamiasz `chmod 644 file`, ustawiasz uprawnienia read/write (odczyt zapis) dla właściciela, uprawnienia read (zapis) dla grupy i read dla wszystkich innych użytkowników. Jeśli skonwertujemy liczbę 644 z systemu ósemkowego na binarny, to otrzymamy `110100100`, lub (w grupach po 3 bity) `110 100 100`.

Teraz widzimy, że każda 'trójka' opisuje uprawnienia dla właściciela/grupy/reszty: pierwsza `rw-`, druga to `r--` i trzecia to `r--`.

System ósemkowy był również popularny na starych komputerach, jak PDP-8, dlatego że słowo (podstawowa porcja informacji) mogło składać się 12, 24 lub 36 bitów, a wszystkie te liczby są podzielne przez 3, więc wybór systemu ósemkowego był całkiem logiczny. Obecnie wszystkie popularne komputery mają słowa/adresy 16-, 32- lub 64-bitowe i wszystkie te liczby są podzielne przez 4, więc system szesnastkowy jest wygodniejszy.

System ósemkowy jest wspierany przez wszystkie standardowe kompilatory C/C++. Czasami jest to źródłem nieporozumień, dlatego że liczby ósemkowe są kodowane z zerem z przodu, na przykład: 0377 to 255. Gdy pomylisz się i napiszesz "09" zamiast 9, to kompilator zgłosi błąd. GCC może zwrócić podobny komunikat :
`error: invalid digit "9" in octal constant`.

System ósemkowy jest również popularny w Javie. Gdy IDA[11] wyświetla string ze znakami niedrukowalnymi, są one zakodowane w systemie ósemkowym (a nie szesnastkowym) . Dekompilator JAD zachowuje się w taki sam sposób.

Podzielność

Kiedy widzisz liczbę 120, to można szybko się zorientować, że jest ona podzielna przez 10, dlatego że ostatnią cyfrą jest 0. Podobnie, 123400 jest podzielne przez 100, bo ostatnie dwie cyfry są zerami.

Analogicznie liczba szesnastkowa 0x1230 jest podzielna przez 0x10 (16 w systemie dziesiętnym), 0x123000 jest podzielne przez 0x1000 (4096 w systemie dziesiętnym), itd.

Liczba binarna 0b1000101000 jest podzielna przez 0b1000 (8), itd.

Tę właściwość można wykorzystać, żeby szybko sprawdzić czy jakiś adres lub rozmiar bloku jest wyrównany do pewnej granicy (czy rozmiar bloku jest całkowitą krotnością długości słowa, np. krotnością 32 bitów). Na przykład sekcje w plikach PE[12] prawie zawsze zaczynają się od adresów kończących się trzema szesnastkowymi zerami: 0x41000, 0x10001000, itd., gdyż prawie wszystkie sekcje w plikach PE są wyrównane do granicy 0x1000 (4096) bajtów.

Arytmetyka wielokrotnej precyzji a podstawa

Arytmetyka wielokrotnej precyzji (multi-precision arithmetic) może operować na dowolnie dużych liczbach, które mogą być przechowywane w kilku bajtach. Na przykład klucze RSA, zarówno prywatne jak i publiczne, mogą zajmować 4096 bitów a nawet więcej.

W [Donald E. Knuth, *The Art of Computer Programming*, Volume 2, 3rd ed., (1997), 265] przedstawiono ideę: kiedy przechowuje się liczbę wielokrotnej precyzji w kilku bajtach, cała liczba może być reprezentowana jako zapisana w systemie liczbowym o podstawie $2^8 = 256$, i każda cyfra reprezentuje jeden bajt. Podobnie, gdybyś liczbę wielokrotnej precyzji przechowywał w kilku 32-bitowych całkowitoliczbowych wartościach, każda cyfra byłaby reprezentowana przez 32-bitowy slot (4 bajty), i można by uważać tę liczbę za zapisaną w systemie o podstawie 2^{32}.

Wymowa liczb w systemach niedziesiętnych

Liczby zapisane w systemie o podstawie innej niż 10, zwykle wymawia się cyfra po cyfrze: “jeden-zero-zero-jeden-jeden-...“. Nie używa się słów jak “dziesięć“ czy “tysiąc“ by przez pomyłkę nie potraktowano liczby jak zapisanej w systemie dziesiętnym.

Liczby zmiennoprzecinkowe

Żeby odróżniać liczby zmiennoprzecinkowe od całkowitoliczbowych, często na końcu dodaje się “.0”, np. 0.0, 123.0, itd.

[11] Interaktywny deasembler i debugger rozwijany przez Hex-Rays
[12] Portable Executable (format plików wykonywalnych w systemach Windows)

1.3 Pusta funkcja

Najprostszą istniejącą funkcją jest funkcja, która nic nie robi:

Listing 1.1: Kod w C/C++

```
void f()
{
        return;
};
```

Skompilujmy ją!

1.3.1 x86

Dla x86 i MSVC, i GCC generują ten sam kod:

Listing 1.2: Optymalizujący GCC/MSVC (wyjście w asemblerze)

```
f:
        ret
```

Mamy tu tylko jedną instrukcję: RET, która jest instrukcją powrotu do funkcji wywołującej.

1.3.2 ARM

Listing 1.3: Optymalizujący Keil 6/2013 (tryb ARM) (wyjście w asemblerze)

```
f       PROC
        BX       lr
        ENDP
```

Adres powrotu (RA[13]) w ARM zapisywany jest nie na stosie, a w rejestrze LR[14]. Instrukcja BX LR, wykonując skok pod ten adres, zwraca sterowanie do funkcji wywołującej.

1.3.3 MIPS

Są dwie konwencje nazewnictwa rejestrów w architekturze MIPS: używająca numeru rejestru (od $0 do $31) lub nazwy umownej ($V0, $A0, itd.).

Wyjście asemblera w GCC pokazuje numery rejestrów

Listing 1.4: Optymalizujący GCC 4.4.5 (wyjście w asemblerze)

```
        j       $31
        nop
```

…a IDA— nazwy umowne:

[13] adres powrotu
[14] Link Register

Listing 1.5: Optymalizujący GCC 4.4.5 (IDA)

```
        j       $ra
        nop
```

Pierwsza instrukcja jest instrukcją skoku (J lub JR), która zwraca sterowanie do funkcji wywołującej, skacząc pod adres w rejestrze $31 ($RA).

Jest to rejestr odpowiadający LR w ARM.

Druga instrukcja to NOP[15], która nic nie robi. Na razie możemy ją zignorować.

Jeszcze małe co nieco o konwencji nazewnictwa w MIPS

Nazwy rejestrów i instrukcji w MIPS tradycyjnie są zapisywane małymi literami, lecz my będziemy je zapisywać dużymi, dlatego że nazwy instrukcji i rejestrów innych ISA w tej książce są zapisywane dużymi.

1.3.4 Puste funkcje w praktyce

Mimo, że puste funkcje są bezużyteczne, są one dość często spotykane w niskopoziomowym kodzie.

Po pierwsze, często spotykamy funkcje zapisujące szczegółowe informacje do logów, na przykład:

Listing 1.6: Kod w C/C++

```
void dbg_print (const char *fmt, ...)
{
#ifdef _DEBUG
        // otwórz plik z logami
        // zapisz do pliku z logami
        // zamknij plik z logami
#endif
};

void some_function()
{
        ...

        dbg_print ("we did something\n");

        ...
};
```

Przy kompilacji wersji programu przeznaczonej do wdrożenia, `_DEBUG` jest niezdefiniowane, więc funkcja `dbg_print()`, mimo, że jest wywoływana, będzie pusta.

Po drugie, popularnym sposobem na ochronę oprogramowania jest kompilacja kilku wersji: pierwsza dla legalnych konsumentów, druga - demonstracyjna. Demonstracyjna wersja może nie zawierać jakiejś ważnej funkcjonalności, na przykład:

[15] No Operation

Listing 1.7: Kod w C/C++

```
void save_file ()
{
#ifndef DEMO
        // kod realizujący zapis
#endif
};
```

Funkcja `save_file()` może być wywołana, kiedy użytkownik klika w menu `File->`Save. Wersja demo może zawierać wyłączony przycisk menu, ale nawet jeśli cracker go włączy, to zostanie wywoływana jedynie pusta funkcja, w której nie ma użytecznego kodu.

IDA oznacza takie funkcje jako `nullsub_00`, `nullsub_01`, itd.

1.4 Zwracanie wartości

Inną prostą funkcją jest taka, która zwraca stałą wartość.

Listing 1.8: Kod w C/C++

```
int f()
{
        return 123;
};
```

Skompilujmy ją.

1.4.1 x86

Poniżej efekt kompilacji z optymalizacją kompilatorami GCC i MSVC na x86:

Listing 1.9: Optymalizujący GCC/MSVC (wyjście w asemblerze)

```
f:
        mov     eax, 123
        ret
```

Są tu tylko dwie instrukcje: pierwsza zapisuje wartość 123 do rejestru EAX, który umownie jest używany do przechowywania wartości zwracanej, a drugą jest RET, która zwraca sterowanie do funkcji wywołującej.

Funkcja wywołująca pobierze wynik z rejestru EAX.

1.4.2 ARM

W kodzie maszynowym na ARM widać kilka różnic:

Listing 1.10: Optymalizujący Keil 6/2013 (tryb ARM) (wyjście w asemblerze)

```
f       PROC
```

```
        MOV     r0,#0x7b ; 123
        BX      lr
        ENDP
```

ARM używa rejestru R0 do zwracania wartości z funkcji, więc 123 kopiowane jest do R0.

Warto zaznaczyć, że nazwa instrukcji MOV jest myląca, zarówno na x86 jak i na ARM.

Dane nie są *przenoszone*, tylko *kopiowane*.

1.4.3 MIPS

Wyjście asemblera GCC oznacza rejestry numerami:

Listing 1.11: Optymalizujący GCC 4.4.5 (wyjście w asemblerze)

```
        j       $31
        li      $2,123                  # 0x7b
```

…a IDA używa nazw umownych:

Listing 1.12: Optymalizujący GCC 4.4.5 (IDA)

```
                jr      $ra
                li      $v0, 0x7B
```

Rejestr $2 (nazwa umowna - $V0) używany jest do przechowywanie wartości zwracanej z funkcji. LI to skrót od “Load Immediate” i w architekturze MIPS jest odpowiednikiem instrukcji MOV z x86.

Drugą instrukcją jest instrukcja skoku (J lub JR), która zwraca sterowanie do funkcji wywołującej.

Możesz się zastanawiać, dlaczego instrukcje LI i J/JR są w odwrotnej kolejności. Jest to efekt optymalizacji przetwarzania potokowego w architekturze RISC, zwanej „branch delay slot”.

Przyczyną wprowadzanie takiego rozwiązania jest dziwactwo w niektórych architekturach typu RISC i nie jest to dla nas istotne - po prostu przyjmijmy, że w asemblerze MIPS instrukcja następująca po instrukcji skoku jump/branch jest wykonywana przed samym skokiem.

W rezultacie instrukcje typu branch są zawsze zamienione z instrukcją, która jest wykonywana przed nimi.

W praktyce często występują funkcje, które jedynie zwracają 1 (*true - prawdę*) lub 0 (*false - fałsz*).

Najmniejsze UNIXowe narzędzia, */bin/true* i */bin/false* zwracają odpowiednio 0 i 1, jako kod wyjścia. (Zero, jako kod wyjścia, oznacza zwykle sukces, natomiast kod niezerowy oznacza błąd.)

1.5 Hello, world!

Przejdźmy do słynnego przykładu z książki [Brian W. Kernighan, Dennis M. Ritchie, *The C Programming Language*, 2ed, (1988)]:

Listing 1.13: Kod w C/C++

```
#include <stdio.h>

int main()
{
    printf("hello, world\n");
    return 0;
}
```

1.5.1 x86

MSVC

Skompilujmy kod w MSVC 2010:

```
cl 1.cpp /Fa1.asm
```

(Opcja /Fa oznacza wygenerowanie listingu w asemblerze)

Listing 1.14: MSVC 2010

```
CONST   SEGMENT
$SG3830 DB      'hello, world', 0AH, 00H
CONST   ENDS
PUBLIC  _main
EXTRN   _printf:PROC
; Function compile flags: /Odtp
_TEXT   SEGMENT
_main   PROC
        push    ebp
        mov     ebp, esp
        push    OFFSET $SG3830
        call    _printf
        add     esp, 4
        xor     eax, eax
        pop     ebp
        ret     0
_main   ENDP
_TEXT   ENDS
```

MSVC generuje listingi w składni Intela. Różnica między składnią Intela a AT&T będzie omówiona w in 1.5.1 on page 15.

Kompilator wygenerował plik `1.obj`, który następnie będzie połączony konsolidatorem (ang. linker) w `1.exe`. W naszym przypadku ten plik składa się z dwóch segmentów: CONST (dane-stałe) i _TEXT (kod).

Łańcuch znaków `hello, world` w C/C++ ma typ `const char[]`[Bjarne Stroustrup, *The C++ Programming Language, 4th Edition*, (2013)p176, 7.3.2], jednak nie posiada nazwy. Kompilator potrzebuje nazwy, żeby z tym łańcuchem znaków pracować, dlatego nadaje mu własną nazwę - `$SG3830`.

Dlatego ten przykład można by zapisać w ten sposób:

```
#include <stdio.h>

const char $SG3830[]="hello, world\n";

int main()
{
    printf($SG3830);
    return 0;
}
```

Wróćmy do kodu w asemblerze. Jak widać, łańcuch znaków kończy się bajtem zerowym — jest to element standardu C/C++ o łańcuchach znaków. Więcej o łańcuchach znaków w C/C++: **??** on page ??.

W segmencie kodu _TEXT znajduje się na razie tylko jedna funkcja: `main()` i, jak prawie każda funkcja, zaczyna się od prologu a kończy się epilogiem [16].

Po prologu następuje wywołanie funkcji `printf()`:
`CALL _printf`. Przed tym wywołaniem adres łańcucha znaków (lub wskaźnik na niego) z naszym pozdrowieniem ("Hello, world!") odkładany jest na stos, za pomocą instrukcji PUSH.

Po tym jak funkcja `printf()` zwraca sterowanie do funkcji `main()`, adres łańcucha znaków (lub wskaźnik na niego) wciąż jest na stosie. Nie jest on dalej potrzebny, więc wskaźnik wierzchołka stosu (rejestr ESP) musi zostać poprawiony.

`ADD ESP, 4` oznacza: dodaj wartość 4 do rejestru ESP.

Dlaczego 4? Z racji tego, że jest to kod 32-bitowy, do przekazania adresu za pomocą stosu potrzebowaliśmy 4 bajtów. W x64 potrzebowalibyśmy 8 bajtów.
`ADD ESP, 4` jest równoważne instrukcji `POP register`, ale nie wykorzystuje dodatkowego rejestru[17]. W pierwszym przypadku jedynie bezpośrednio manipulujemy na rejestrze ESP (wskaźniku wierzchołka stosu), a w drugim przypadku zdejmujemy ze stosu jeden element i odkładamy go do rejestru *register* a rejestr ESP zmienia się automatycznie.

Niektóre kompilatory, jak Intel C++ Compiler, w tej samej sytuacji mogą wygenerować `POP ECX` zamiast `ADD` (można to zaobserwować w kodzie Oracle RDBMS, gdyż jest on kompilowany właśnie tym kompilatorem). `POP ECX` oznacza zdjęcie elementu ze stosu i umieszczenie go w rejestrze ECX. Osiągnęliśmy taki sam efekt jak w przypadku instrukcji `ADD` (zmiana wskaźnika stosu), ale skutkiem ubocznym jest nadpisanie ECX.

Możliwe, że kompilator stosuje tu `POP ECX` dlatego, że ta instrukcja jest krótsza (1 bajt w przypadku `POP` kontra 3 bajty w przypadku `ADD`).

[16] Więcej o prologu i epilogu przeczytasz w sekcji (1.6 on page 40).
[17] Ale za to modyfikowany jest rejestr stanu

Poniżej przykład wykorzystania POP zamiast ADD z Oracle RDBMS:

Listing 1.15: Oracle RDBMS 10.2 Linux (plik app.o)

```
.text:0800029A                 push    ebx
.text:0800029B                 call    qksfroChild
.text:080002A0                 pop     ecx
```

MSVC czasami zachowuje się tak samo.

Listing 1.16: Saper na systemie Windows 7 32-bit

```
.text:0102106F                 push    0
.text:01021071                 call    ds:time
.text:01021077                 pop     ecx
```

Po wywołaniu `printf()` kod w C/C++ zawiera instrukcję `return 0` — zwróć 0 jako wynik funcji `main()`.

W kodzie wygenerowanym robi to instrukcja `XOR EAX, EAX`.

XOR, jak można się domyśleć, to — „alternatywa wykluczająca"[18], ale kompilatory często korzystają z niej zamiast z `MOV EAX, 0` — znów dlatego, że kod operacji (opcode) jest krótszy (2 bajty w XOR kontra 5 w MOV).

Niektóre kompilatory generują `SUB EAX, EAX`, co oznacza *odejmij wartość w* EAX *od wartości w* EAX, co w każdym przypadku da 0.

Ostatnia instrukcja RET zwraca sterowanie do funkcji wywołującej. Zwykle jest to kod C/C++ CRT[19], który z kolei zwróci sterowanie do systemu operacyjnego.

GCC

Skompilujmy teraz ten sam kod za pomocą kompilatora GCC 4.4.1 na systemie Linux: `gcc 1.c -o 1`. Następnie za pomocą deasemblera IDA podejrzymy wynik kompilacji funkcji `main()`. IDA, jak i MSVC, pokazują kod w składni Intela[20].

Listing 1.17: Kod w programie IDA

```
main            proc near

var_10          = dword ptr -10h

                push    ebp
                mov     ebp, esp
                and     esp, 0FFFFFFF0h
                sub     esp, 10h
                mov     eax, offset aHelloWorld ; "hello, world\n"
                mov     [esp+10h+var_10], eax
                call    _printf
                mov     eax, 0
```

[18] Wikipedia
[19] C Runtime library
[20] Można zmusić również GCC do generowania listingów w tym formacie, za pomocą opcji `-S -masm=intel`.

```
                leave
                retn
main            endp
```

Wynik jest prawie taki sam. Adres łańcucha znaków `hello, world`, leżącego w segmencie danych, najpierw zapisywany jest do EAX, a później odkładany na stos. Dodatkowo w prologu funkcji widzimy `AND ESP, 0FFFFFFF0h` — ta instrukcja wyrównuje ESP do granicy 16 bajtów. Dzięki temu wszystkie wartości na stosie będą również wyrównane w taki sam sposób (procesor pracuje efektywniej z adresami wyrównanymi do granicy 4 lub 16 bajtów.).

`SUB ESP, 10h` alokuje na stosie 16 bajtów. Choć tutaj wystarczyłyby 4 bajty, co będzie widoczne dalej.

Dzieje się tak dlatego, że ilość przydzielanego miejsca na stosie jest również wyrównywana do granicy 16 bajtów.

Adres łańcucha znaków (lub wskaźnik na niego) jest zatem odkładany prosto na stos, bez wykorzystywania instrukcji PUSH. *var_10* jest jednocześnie zmienną lokalną i argumentem dla funkcji `printf()`. Więcej na ten temat dowiesz się później.

Następnie jest wywoływana funkcja `printf()`.

W odróżnieniu od MSVC, GCC przy kompilacji z wyłączoną optymalizacją generuje `MOV EAX, 0` zamiast krótszego kod operacji (opcode).

Ostatnia instrukcja LEAVE — jest analogiczna do pary instrukcji `MOV ESP, EBP` i `POP EBP` — jest to powrót wskaźnika stosu i rejestru EBP do stanu początkowego. Jest to niezbędne, ponieważ na początku funkcji modyfikowaliśmy rejestry ESP i EBP (za pomocą `MOV EBP, ESP` / `AND ESP, …`).

GCC: składnia AT&T

Sprawdźmy, jak będzie wyglądał listing w składni AT&T, która jest bardziej popularna w świecie UNIXa.

Listing 1.18: Kompilujemy za pomocą GCC 4.7.3

```
gcc -S 1_1.c
```

Otrzymamy taki plik:

Listing 1.19: GCC 4.7.3

```
        .file   "1_1.c"
        .section        .rodata
.LC0:
        .string "hello, world\n"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        pushl   %ebp
```

```
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        movl    %esp, %ebp
        .cfi_def_cfa_register 5
        andl    $-16, %esp
        subl    $16, %esp
        movl    $.LC0, (%esp)
        call    printf
        movl    $0, %eax
        leave
        .cfi_restore 5
        .cfi_def_cfa 4, 4
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .ident  "GCC: (Ubuntu/Linaro 4.7.3-1ubuntu1) 4.7.3"
        .section        .note.GNU-stack,"",@progbits
```

Listing zawiera wiele makr (rozpoczynających się od kropki). Na razie nie są one dla nas istotne.

Zignorujmy je wszystkie, (za wyjątkiem *.string*, które koduje sekwencję znaków zakończonych znakiem null — tak jak łańcuchy znaków w C++). Otrzymamy wtedy: [21]:

Listing 1.20: GCC 4.7.3

```
.LC0:
        .string "hello, world\n"
main:
        pushl   %ebp
        movl    %esp, %ebp
        andl    $-16, %esp
        subl    $16, %esp
        movl    $.LC0, (%esp)
        call    printf
        movl    $0, %eax
        leave
        ret
```

Główne różnice między składnią Intela a AT&T :

- Operandy są zapisywane w odwotnej kolejności

  W składni Intela:
  <instrukcja> <operand docelowy> <operand źródłowy>.

  W składni AT&T:
  <instrukcja> <operand źródłowy> <operand docelowy>.

  Istnieje łatwy sposób na zapamiętanie tej różnicy: kiedy pracujecie ze składnią Intela — możecie w głowie postawić znak równości (=) między operandami, a

[21] Eliminację zbędnych makr można uzyskać za pomocą opcji GCC: *-fno-asynchronous-unwind-tables*

z AT&T — strzałkę w prawo (→) [22].

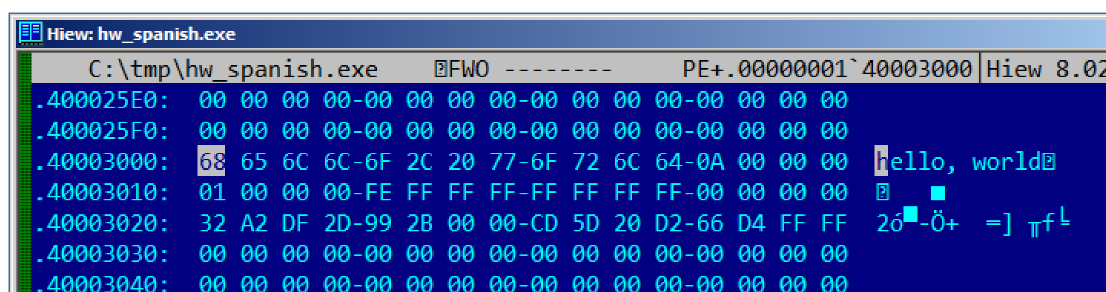
- AT&T: Przed nazwami rejestrów stawia się symbol (%), a przed liczbami ($). Zamiast nawiasów kwadratowych używa się okrągłych.
- AT&T: Do każdej instrukcji dodaje się przyrostek określający typ danych:
  - q — quad (64 bity)
  - l — long (32 bity)
  - w — word (16 bitów)
  - b — byte (8 bitów)

Wracając do wyniku kompilacji: jest on niemal identyczny do tego, który prezentuje IDA. Jedna drobnostka: 0FFFFFFF0h jest zapisywane jako $-16. Oba zapisy oznaczją dokładnie to samo: 16 w sytemie dziesiętnym to 0x10 w szesnastkowym. -0x10 to 0xFFFFFFF0 (dla liczb całkowitych 32-bitowych) [23].

Zwracana wartość jest ustawiany na 0 za pomocą zwykłej instrukcji MOV, a nie XOR. MOV zapisuje wartość do rejestru. Nazwa tej instrukcji nie jest do końca poprawna (wartości nie są przemieszczane, tylko kopiowane). W innych architekturach instrukcja ta nosi nazwę „LOAD", „STORE" lub podobną.

Korekcja (patching) łańcuchów znaków (Win32)

Możemy w łatwy sposób znaleźć łańcuch znaków „hello, world" w pliku wykonywalnym za pomocą Hiew:

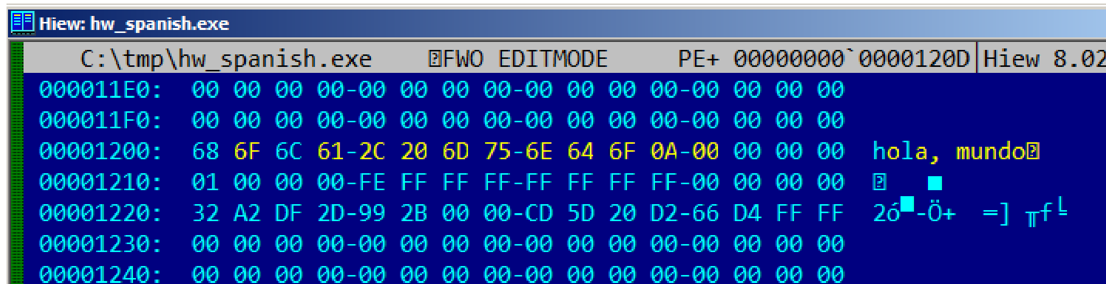


Rysunek 1.1: Hiew

Możemy przetłumaczyć naszą wiadomość na język hiszpański:

[22] W niektórych standardowych funkcjach biblioteki C (memcpy(), strcpy(), ...) również korzysta się z kolejności argumentów jak w składni Intela: najpierw wskaźnik na miejsce docelowe w pamięci, następnie wskaźnik na miejsce źródłowe.

[23] W kodowaniu U2 - https://pl.wikipedia.org/wiki/Kod_uzupe%C5%82nie%C5%84_do_dw%C3%B3ch i kolejności bajtów big endian - https://pl.wikipedia.org/wiki/Kolejno%C5%9B%C4%87_bajt%C3%B3w



Rysunek 1.2: Hiew

Tekst w języku hiszpańskim jest o 1 bajt krótszy od tekstu w języku angielskim, dlatego dodajemy na koniec bajt 0x0A (\n) i bajt zerowy.

Działa.

A co jeśli chcielibyśmy wstawić dłuższy tekst? Po oryginalnym tekście w języku angielskim widzimy kilka bajtów zerowych. Trudno powiedzieć czy można je nadpisać: mogą (ale nie muszą!) one być wykorzystywane gdzieś w kodzie CRT. Tak czy inaczej, możemy je nadpisywać tylko jeśli naprawdę wiemy co robimy.

Korekcja łańcuchów znaków (Linux x64)

Spróbujmy edytować plik wykonywalny systemu Linux x64, korzystając z rada.re:

Listing 1.21: Sesja w rada.re

```
dennis@bigbox ~/tmp % gcc hw.c

dennis@bigbox ~/tmp % radare2 a.out
 -- SHALL WE PLAY A GAME?
[0x00400430]> / hello
Searching 5 bytes from 0x00400000 to 0x00601040: 68 65 6c 6c 6f
Searching 5 bytes in [0x400000-0x601040]
hits: 1
0x004005c4 hit0_0 .HHhello, world;0.

[0x00400430]> s 0x004005c4

[0x004005c4]> px
- offset -   0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0123456789ABCDEF
0x004005c4  6865 6c6c 6f2c 2077 6f72 6c64 0000 0000  hello, world....
0x004005d4  011b 033b 3000 0000 0500 0000 1cfe ffff  ...;0...........
0x004005e4  7c00 0000 5cfe ffff 4c00 0000 52ff ffff  |...\...L...R...
0x004005f4  a400 0000 6cff ffff c400 0000 dcff ffff  ....l...........
0x00400604  0c01 0000 1400 0000 0000 0000 017a 5200  .............zR.
0x00400614  0178 1001 1b0c 0708 9001 0710 1400 0000  .x..............
0x00400624  1c00 0000 08fe ffff 2a00 0000 0000 0000  ........*.......
0x00400634  0000 0000 1400 0000 0000 0000 017a 5200  .............zR.
0x00400644  0178 1001 1b0c 0708 9001 0000 2400 0000  .x..........$...
```

```
0x00400654  1c00 0000 98fd ffff 3000 0000 000e 1046  ........0......F
0x00400664  0e18 4a0f 0b77 0880 003f 1a3b 2a33 2422  ..J..w...?.;*3$"
0x00400674  0000 0000 1c00 0000 4400 0000 a6fe ffff  ........D.......
0x00400684  1500 0000 0041 0e10 8602 430d 0650 0c07  .....A....C..P..
0x00400694  0800 0000 4400 0000 6400 0000 a0fe ffff  ....D...d.......
0x004006a4  6500 0000 0042 0e10 8f02 420e 188e 0345  e....B....B....E
0x004006b4  0e20 8d04 420e 288c 0548 0e30 8606 480e  . ..B.(..H.0..H.

[0x004005c4]> oo+
File a.out reopened in read-write mode

[0x004005c4]> w hola, mundo\x00

[0x004005c4]> q

dennis@bigbox ~/tmp % ./a.out
hola, mundo
```

Co tu się dzieje: szukam łańcucha znaków „hello”, korzystając z komendy `/`, następnie ustawiam *kursor* (*seek* w terminologii rada.re) pod ten adres. Następnie chcę się upewnić, że jest to rzeczywiście poszukiwane miejsce: `px` wyświetla bajty pod tym adresem. `oo+` przełącza rada.re w tryb *odczytu/zapis*. `w` zapisuje łańuch znaków ASCII w miejscu kursora (*seek*). Warto zauważyć `\00` na końcu, jest to bajt zerowy. `q` wyłącza rada.re.

Prawdziwa historia crackowania oprogramowania

Program do przetwarzania obrazów, niezarejestrowany, dodawał do pliku znak wodny, na przykład napis „This image was processed by evaluation version of [nazwa oprogramowania]” Spróbowaliśmy najprostszego rozwiązania: znaleźliśmy ten tekst w pliku wykonywalnym i zastąpiliśmy go spacjami. Znak wodny zniknął. Ogólnie rzecz biorąc, wciąż był nakładany przez program. Za pomocą funkcji Qt, znak wodny wciąż był dodawany do obrazu. Ale dodawanie spacji nie zmieniało go w żaden sposób...

Tłumaczenie oprogramowania za czasów MS-DOS

Sposób przedstawiony wyżej był powszechnie wykorzystywany w latach 80. i 90. przy tłumaczeniu oprogramowania pod MS-DOS na język rosyjski. Ta technika może być wykorzystywana przy braku wiedzy na temat kodu maszynowego i formatów plików wykonywalnych. Nowy łańcuch znaków nie powinien być dłuższy niż stary, ponieważ istnieje ryzyko nadpisania innej wartości albo fragmentu kodu wykonywalnego

Rosyjskie słowa i zdania zwykle są trochę dłuższe od angielskich odpowiedników, dlatego *przetłumaczone* oprogramowanie zawierało sporo dziwnych akronimów (skrótowców) i trudnych do zrozumienia skrótów.

Rysunek 1.3: Norton Commander 5.51 przetłumaczony na język rosyjski

Prawdopodobnie sytuacja wyglądała podobnie z tłumaczeniem na inne języki.

W przypadku łańcuch znaków w Delphi, rozmiar również musi być poprawiony, jeśli zachodzi taka potrzeba.

1.5.2 x86-64

MSVC: x86-64

Przyjrzyjmy się wynikom kompilacji 64-bitowego MSVC:

Listing 1.22: MSVC 2012 x64

```
$SG2989 DB      'hello, world', 0AH, 00H

main    PROC
        sub     rsp, 40
        lea     rcx, OFFSET FLAT:$SG2989
        call    printf
        xor     eax, eax
        add     rsp, 40
        ret     0
main    ENDP
```

W x86-64 wszystkie rejestry zostały rozszerzone do 64 bitów i ich nazwy zyskały prefiks R-. Żeby jak najrzadziej korzystać ze stosu (inaczej mówiąc, jak najmniej korzystać z pamięci cache i pamięci zewnętrznej), istnieje popularna metoda przekazywania argumentów funkcji przez rejestry (*fastcall*) **??** on page ??. Tzn. część argumentów funkcji jest przekazywana przez rejestry a część — przez stos. W Win64 pierwsze 4 argumenty funkcji są przekazywane przez rejestry RCX, RDX, R8 i R9. Widać to w powyższym przykładzie: wskaźnik na argument funkcji `printf()` (łańcuch znaków) teraz jest przekazywany nie przez stos, a przez rejestr RCX. Wskaźniki są teraz 64-bitowe, więc są przekazywane przez przez 64-bitowe rejestry (mające prefiks R-). Ale dla wstecznej kompatybilności można adresować również młodsze 32 bity rejestrów poprzez prefiks E-. W ten oto sposób wygląda rejestr RAX/EAX/AX/AL w x86-64:

Number bajtu:							
7	6	5	4	3	2	1	0
RAX^{x64}							
				EAX			
						AX	
						AH	AL

Funkcja `main()` zwraca wartość typu *int*, który w C/C++, dla większej kompatybilności, pozostał 32-bitowy. Właśnie dlatego na końcu funkcji `main()` zeruje się nie RAX, a EAX, czyli 32-bitową część rejestru. Dodatkowo na stosie lokalnym jest zarezerwowanych 40 bajtów. Jest to tzw. „shadow space", który będzie omawiany później: 1.14.2 on page 134.

GCC: x86-64

Przyjrzyjmy się wynikom kompilacji GCC na 64-bitowym systemie Linux:

Listing 1.23: GCC 4.4.6 x64

```
.string "hello, world\n"
main:
        sub     rsp, 8
        mov     edi, OFFSET FLAT:.LC0 ; "hello, world\n"
        xor     eax, eax  ; liczba użytych rejestrów wektorowych XMM0-XMM7
        call    printf
        xor     eax, eax
        add     rsp, 8
        ret
```

Na Linuksie, *BSD i Mac OS X w architekturze x86-64 argumenty funkcji także przekazuje się przez rejestry [Michael Matz, Jan Hubicka, Andreas Jaeger, Mark Mitchell, *System V Application Binary Interface. AMD64 Architecture Processor Supplement*, (2013)] [24].

6 pierwszych argumentów jest przekazywanych przez rejestry RDI, RSI, RDX, RCX, R8 i R9, a reszta — przez stos.

Wskaźnik na łańcuch znaków jest przekazywany przez EDI (32-bitową część rejestru). Dlaczego nie użyto 64-bitowego RDI?

Warto pamiętać, że w 64-bitowym trybie wszystkie instrukcje MOV, zapisujące cokolwiek do młodszej 32-bitowej części rejestru, zerują starsze 32 bity (jest to opisane w dokumentacji Intela: 7.1.4 on page 150). Z tego powodu instrukcja `MOV EAX, 011223344h` poprawnie zapisze tę wartość do RAX, a starsze bity się wyzerują.

Jeślibyśmy podejrzeli w deasemblerze IDA skompilowany plik (.o), to zobaczylibyśmy kody operacji (opcode) wszystkich instrukcji [25]:

Listing 1.24: GCC 4.4.6 x64

```
.text:00000000004004D0                   main  proc near
.text:00000000004004D0 48 83 EC 08       sub     rsp, 8
.text:00000000004004D4 BF E8 05 40 00    mov     edi, offset format ; "hello,
    world\n"
.text:00000000004004D9 31 C0             xor     eax, eax
.text:00000000004004DB E8 D8 FE FF FF    call    _printf
.text:00000000004004E0 31 C0             xor     eax, eax
.text:00000000004004E2 48 83 C4 08       add     rsp, 8
```

[24] Dostęp także przez https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf

[25] Trzeba włączyć tę opcję w **Options** → **Disassembly** → **Number of opcode bytes**

```
.text:00000000004004E6 C3                   retn
.text:00000000004004E6                      main  endp
```

Jak widać, instrukcja MOV pod adresem 0x4004D4, która zapisuje do EDI, zajmuje 5 bajtów. Ta sama instrukcja, zapisująca 64-bitową wartość do RDI, zajmuje 7 bajtów. Najwyraźniej GCC stara się zaoszczędzić trochę miejsca. GCC jest również pewne, że segment danych, w którym przechowywany jest łańcuch znaków, nigdy nie będzie zaalokowany pod adresem powyżej 4GiB.

Widać również, że rejestr EAX został wyzerowany przed wywołaniem funkcji printf(). Zgodnie ze standardem ABI[26] opisanym wyżej, w *NIX dla x86-64 w EAX jest ustawiana liczba użytych rejestrów wektorowych do przekazania argumentów zmiennoprzecinkowych, jeśli funkcja może przyjmować zmienną liczbę argumentów. Nie został wykorzystany żaden taki rejestr, a printf() jest funkcją o zmiennej liczbie argumentów, więc należy ustawić EAX na 0.

Łatanie (patching) adresów (Win64)

Jeśli nasz przykład skompilujemy za pomocą MSVC 2013 z opcją /MD (dynamiczne linkowanie, mniejszy plik wykonywalny dzięki zewnętrznym odwoływaniom do bibliotek MSVCR*.DLL), funkcja main() będzie łatwa do znalezienia, gdyż wystąpi jako pierwsza:

[26] Interfejs binarny aplikacji (Application Binary Interface)

```
Hiew: hw2.exe
   C:\tmp\hw2.exe              FWO EDITMODE a64 PE+ 00000000`00000404 Hiew 8.02 (c)SEN
 00000400: 4883EC28                          sub         rsp,028 ;'('
 00000404: 488D0DF51F0000                    lea         rcx,[000002400]
 0000040B: FF15D7100000                      call        q,[0000014E8]
 00000411: 33C0                              xor         eax,eax
 00000413: 4883C428                          add         rsp,028 ;'('
 00000417: C3                                retn ; -^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-
 00000418: 4883EC28                          sub         rsp,028 ;'('
 0000041C: B84D5A0000                        mov         eax,000005A4D ;'  ZM'
 00000421: 663905D8EFFFFF                    cmp         [-000000C00],ax
 00000428: 7404                              jz          00000042E
 000004
 000004    lea         rcx,[0000000000002401]
 000004                  CommandSelect: Off
 000004
 000004
 0000043F: 813850450000                      cmp         d,[rax],000004550 ;'  EP'
 00000445: 75E3                              jnz         00000042A
 00000447: B90B020000                        mov         ecx,00000020B
 0000044C: 66394818                          cmp         [rax][018],cx
 00000450: 75D8                              jnz         00000042A
 00000452: 33C9                              xor         ecx,ecx
 00000454: 83B8840000000E                    cmp         d,[rax][000000084],00E
 0000045B: 7609                              jbe         000000466
 0000045D: 3988F8000000                      cmp         [rax][0000000F8],ecx
 1Help   2      3      4Select 5      6      7      8      9      10     11
```

Rysunek 1.4: Hiew

Możemy spróbować inkrementować adres:

```
Hiew: hw2.exe
   C:\tmp\hw2.exe          FUO -------- a64 PE+.00000001`4000100B Hiew 8.02 (c)SEN
.40001000: 4883EC28                     sub         rsp,028 ;'('
.40001004: 488D0DF61F0000               lea         rcx,[00000001`40003001] ;'ello, w
.4000100B: FF15D7100000                 call        printf
.40001011: 33C0                         xor         eax,eax
.40001013: 4883C428                     add         rsp,028 ;'('
.40001017: C3                           retn ; -^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-
.40001018: 4883EC28                     sub         rsp,028 ;'('
.4000101C: B84D5A0000                   mov         eax,000005A4D ;'  ZM'
.40001021: 663905D8EFFFFF               cmp         [00000001`40000000],ax
.40001028: 7404                         jz         .00000001`4000102E --2
.4000102A: 33C9                        5xor         ecx,ecx
.4000102C: EB38                         jmps       .00000001`40001066 --3
.4000102E: 48630507F0FFFF              2movsxd      rax,d,[00000001`4000003C] --4
.40001035: 488D0DC4EFFFFF               lea         rcx,[00000001`40000000]
.4000103C: 4803C1                       add         rax,rcx
.4000103F: 813850450000                 cmp         d,[rax],000004550 ;'  EP'
.40001045: 75E3                         jnz        .00000001`4000102A --5
.40001047: B90B020000                   mov         ecx,00000020B
.4000104C: 66394818                     cmp         [rax][018],cx
.40001050: 75D8                         jnz        .00000001`4000102A --5
.40001052: 33C9                         xor         ecx,ecx
.40001054: 83B8840000000E               cmp         d,[rax][000000084],00E
.4000105B: 7609                         jbe        .00000001`40001066 --3
.4000105D: 3988F8000000                 cmp         [rax][0000000F8],ecx
1Help  2PutBlk 3Edit  4Mode  5Goto  6Refer  7Search 8Header 9Files 10Quit 11Hem
```

Rysunek 1.5: Hiew

Hiew wyświetla teraz „ello, world" (obok instrukcji 'LEA RCX...', która ładuje adres łańcucha znaków, przekazywany jako argument do funkcji `printf()`). Kiedy uruchomimy plik wykonywalny, właśnie ten ciąg znaków zostnie wypisany na ekran.

Wyświetlanie różnych ciągów znaków z pliku wykonywalnego (Linux x64)

Plik wykonywalny po skompilowaniu za pomocą GCC 5.4.0 na systemie Linux x64, ma wiele innych łańcuchów znaków: głównie są to nazwy importowanych funkcji oraz nazwy bibliotek.

Uruchamiamy objdump, żeby podejrzeć zawartość wszystkich sekcji skompilowanego pliku:

```
% objdump -s a.out

a.out:     file format elf64-x86-64

Contents of section .interp:
 400238 2f6c6962 36342f6c 642d6c69 6e75782d  /lib64/ld-linux-
 400248 7838362d 36342e73 6f2e3200           x86-64.so.2.
Contents of section .note.ABI-tag:
 400254 04000000 10000000 01000000 474e5500  ............GNU.
```

```
 400264 00000000 02000000 06000000 20000000  ............ ...
Contents of section .note.gnu.build-id:
 400274 04000000 14000000 03000000 474e5500  ............GNU.
 400284 fe461178 5bb710b4 bbf2aca8 5ec1ec10  .F.x[.......^...
 400294 cf3f7ae4                             .?z.

...
```

Łatwo przekazać adres łańcucha znaków „/lib64/ld-linux-x86-64.so.2” do funkcji `printf()`:

```
#include <stdio.h>

int main()
{
    printf(0x400238);
    return 0;
}
```

Trudno uwierzyć, ale program wyświetli ten łańcuch znaków na ekran.

Jeśli zmienimy adres na `0x400260`, to wyświetli się napis „GNU”. Adres jest prawidłowy dla konkretnej wersji GCC, GNU toolset, etc. W waszym systemie plik wykonywalny może wyglądać trochę inaczej i wszystkie adresy także będą inne. Podobnie, usuwanie lub dodawanie kodu do kodu źródłowego może przesunąć wszystkie adresy w programie wykonywalnym do przodu lub do tyłu.

1.5.3 ARM

Do eksperymentów z ARM skorzystamy z kilku kompilatorów:

- popularnego w systemach wbudowanych Keil Release 6/2013,
- Apple Xcode 4.6.3 IDE z kompilatorem LLVM-GCC 4.2 [27],
- GCC 4.9 (Linaro) (dla ARM64), jest dostępny w postaci pliku wykonywalnego dla win32 na `http://www.linaro.org/projects/armv8/`.

Wszędzie w tej książce, jeżeli zaznaczono inaczej, mówimy o 32-bitowym ARM (włączając tryb Thumb i Thumb-2). 64-bitowym ARM będzie oznaczony explicite jako ARM64.

Nieoptymalizujący Keil 6/2013 (tryb ARM)

Na początek skompilujmy nasz przykład za pomocą Keil:

```
armcc.exe --arm --c90 -O0 1.c
```

Kompilator *armcc* generuje listing w asemblerze w składni Intela. Listing zawiera niektóre wysokopoziomowe makra, związane z ARM [28]. Nas interesują prawdziwe instrukcje, dlatego zobaczmy jak wygląda skompilowany kod w programie IDA.

[27] W rzeczywistości Apple Xcode 4.6.3 korzysta z GCC jako front-endu i z LLVM jako generatora kodu binarnego

[28] Na przykład listing zawiera instrukcję PUSH/POP, których nie ma w trybie ARM

Listing 1.25: Nieoptymalizujący Keil 6/2013 (tryb ARM) IDA

```
.text:00000000             main
.text:00000000 10 40 2D E9    STMFD   SP!, {R4,LR}
.text:00000004 1E 0E 8F E2    ADR     R0, aHelloWorld ; "hello, world"
.text:00000008 15 19 00 EB    BL      __2printf
.text:0000000C 00 00 A0 E3    MOV     R0, #0
.text:00000010 10 80 BD E8    LDMFD   SP!, {R4,PC}

.text:000001EC 68 65 6C 6C+aHelloWorld  DCB "hello, world",0    ; DATA XREF:
    main+4
```

Widać, że każda instrukcja ma rozmiar 4 bajtów — zgodnie z oczekiwaniami, ponieważ kompilowaliśmy nasz kod dla trybu ARM, a nie Thumb.

Pierwsza instrukcja, STMFD SP!, {R4,LR}[29], działa podobnie jak instrukcja PUSH w x86: odkłada wartości dwóch rejestrów (R4 i LR) na stos.

W rzeczy samej, kompilator *armcc*, generując listing, wstawił tam dla uproszczenia instrukcję PUSH {r4,lr}. Nie jest to do końca precyzyjne, ponieważ instrukcja PUSH dostępna jest w trybie Thumb. By uniknąć dezorientacji, wygenerowany kod maszynowy podglądamy w programie IDA.

Instrukcja najpierw zmniejsza SP[31], by wskazywał na miejsce na stosie dostępne do zapisu nowych wartości, następnie zapisuje wartości rejestrów R4 i LR pod adres w pamięci, na który wskazuje zmodyfikowany rejestr SP.

Podobnie jak PUSH w trybie Thumb, ta instrukcja pozwala na odkładanie na stos wartości kilku rejestrów na raz, co może być bardzo wygodne.

Przy okazji, takie zachowanie nie ma swojego odpowiednika w x86.

Można zauważyć, że STMFD jest generalizacją instrukcji PUSH (czyli rozszerza jej możliwości), dlatego że może operować na różnych rejestrach, a nie tylko na SP. Inaczej mówiąc, z STMFD można korzystać przy zapisie wartości kilku rejestrów we wskazane miejsce w pamięci.

Instrukcja ADR R0, aHelloWorld dodaje/odejmuje wartość w rejestrze PC[32] (R0) do-/od przesunięcia, w którym jest przechowywany łańcuch znaków hello, world. Dlaczego użyto tutaj rejestru PC? Jest to tzw. „position-independent code” [33].

Taki kod można uruchomić z dowolnego miejsca w pamięci. Inaczej mówiąc, jest to adresowanie względne, względem rejestru PC. W kodzie operacji (opcode) instrukcji ADR jest zapisane przesunięcie (offset) między adresem tej instrukcji a adresem łańcucha znaków. Przesunięcie zawsze jest stałe, niezależnie od tego, w które miejsce OS załadował nasz kod. Dlatego wszystko czego potrzebujemy — to dodanie adresu bieżącej instrukcji (z PC), żeby otrzymać adres bezwględny łańcucha znaków.

Instrukcja BL __2printf[34] wywołuje funkcję printf(). Działanie tej instrukcji przebiega w 2 krokach:

[29] STMFD[30]
[31] wskaźnik stosu. SP/ESP/RSP w x86/x64. SP w ARM.
[32] Program Counter. IP/EIP/RIP w x86/64. PC w ARM.
[33] Jest to szerzej omówione w kolejnym rozdziale (**??** on page ??)
[34] Branch with Link

- zapisz adres występujący po instrukcji BL (`0xC`) do rejestru LR,
- przekaż sterowanie do funkcji `printf()`, zapisując jej adres do rejestru PC.

Kiedy funkcja `printf()` zakończy działanie, musi wiedzieć gdzie zwrócić sterowanie. Dlatego każda funkcja, kończąc pracę, zwraca sterowanie pod adres zapisany w rejestrze LR.

Na tym polega główna różnica między „czystymi" procesorami RISC, jak ARM, a procesorami CISC[35] w rodzaju x86, gdzie adres powrotu zwykle jest odkładany na stos. Przeczytasz o tym więcej w kolejnym rozdziale (1.9 on page 42).

Dodatkowo nie jest możliwe zakodowanie 32-bitowego adresu bezwzględnego (lub przesunięcia) w 32-bitowej instrukcji BL, ponieważ ma ona miejsce tylko dla 24 bitów. Jak zapewne pamiętasz, wszystkie instrukcje w trybie ARM mają długość 4 bajtów (32 bitów) i mogą się znajdować tylko pod adresem wyrównanym do krotności 4 bajtów. Oznacza to, że ostatnich 2 bitów (które są zawsze zerowe) można nie kodować. Ostatecznie zostaje nam 26 bitów na zakodowanie przesunięcia. Odpowiada to zakresowi $current_PC \pm \approx 32M$.

Następna instrukcja `MOV R0, #0`[36] po prostu zapisuje 0 do rejestru `R0`. To dlatego, że nasza funkcja zwraca 0, a wartości zwracane z funkcji zapisywane są do `R0`.

Ostatnia instrukcja to `LDMFD SP!, R4,PC`[37]. Pobiera ona wartość ze stosu (lub z pamięci), zapisuje do R4 i PC oraz zwiększa wskaźnik stosu SP. Działa podobnie jak instrukcja `POP`.
Notabene, pierwsza instrukcja `STMFD` odłożyła na stos wartości z rejestrów R4 i LR, ale teraz zostały one przywrócone do R4 i PC, dzięki instrukcji `LDMFD`.

Jak już wiemy, rejestr LR zawiera adres w pamięci, pod który funkcja zwróci sterowanie po zakończeniu swojej pracy. Pierwsza instrukcja odkłada tę wartość na stos, ponieważ ten sam rejestr zostanie wykorzystany przez naszą funkcję `main()`, gdy wywoła ona funkcję `printf()`.

Na końcu funkcji ta wartość zapisywana jest do rejestru PC, w ten sposób przekazując sterowanie tam, skąd została wywołana.

Z reguły funkcja `main()` jest funkcją główną w C/C++, więc zarządzanie zostanie zwrócone do loadera w OS, lub gdzieś do CRT, lub w jeszcze inne, podobne, miejsce.

Wszystko to pozwala pozbyć się ręcznego wywoływania `BX LR` (skok pod adres z rejestru LR) na samym końcu funcji.

`DCB` — dyrektywa asemblera, opisująca tablicę bajtów bądź ciąg znaków ASCII, podobna do dyrektywy DB w x86

Nieoptymalizujący Keil 6/2013 (tryb Thumb)

Skompilujmy ten sam przykład za pomocą Keil w trybie Thumb:

```
armcc.exe --thumb --c90 -O0 1.c
```

[35] Complex Instruction Set Computing
[36] Oznacza MOV
[37] LDMFD[38] jest instrukcją odwrotną do STMFD

Otrzymamy (listing z programu IDA):

Listing 1.26: Nieoptymalizujący Keil 6/2013 (tryb Thumb) + IDA

```
.text:00000000             main
.text:00000000 10 B5          PUSH    {R4,LR}
.text:00000002 C0 A0          ADR     R0, aHelloWorld ; "hello, world"
.text:00000004 06 F0 2E F9    BL      __2printf
.text:00000008 00 20          MOVS    R0, #0
.text:0000000A 10 BD          POP     {R4,PC}

.text:00000304 68 65 6C 6C+aHelloWorld  DCB "hello, world",0    ; DATA XREF:
    main+2
```

Od razu można zauważyć 2 bajtowe (16-bitowe) kody operacji (opcode) — jest to już wcześniej wspomniany tryb Thumb.

Wyjątkiem jest instrukcja BL, która składa się z dwóch 16-bitowych instrukcji. W jednym 16-bitowym kodzie operacji (opcode) jest za mało miejsca na przesunięcie, po którym znajduje się funkcja `printf()`. Dlatego pierwsza 16-bitowa instrukcja ładuje starsze 10 bitów przesunięcia, a druga — młodsze 11 bitów przesunięcia.

Skoro wszystkie instrukcje w trybie Thumb są 2-bajtowe (16-bitowe), to sytuacja, w której instrukcja zaczyna się pod adresem nieparzystym, jest niemożliwa.

Wniosek z tego jest taki, że ostatniego bitu adresu można nie kodować. Dzięki temu instrukcja BL w trybie Thumb może zakodować adres z przedziału $current_PC \pm \approx 2M$.

Co do pozostałych instrukcji: PUSH i POP działają jak opisane wyżej STMFD/LDMFD, tyle że rejestr SP nie jest tu jawnie wskazany. ADR działa dokładnie tak samo jak w poprzednim przykładzie. MOVS zapisuje wartość 0 do rejestru R0, by funkcja zwróciła zero.

Optymalizujący Xcode 4.6.3 (LLVM) (tryb ARM)

Xcode 4.6.3 bez włączonego trybu optymalizacji generuje za dużo zbędnego kodu, dlatego włączymy optymalizację (flaga -O3), dzięki czemu liczba instrukcji będzie tak mała, jak to możliwe.

Listing 1.27: Optymalizujący Xcode 4.6.3 (LLVM) (tryb ARM)

```
__text:000028C4             _hello_world
__text:000028C4 80 40 2D E9   STMFD           SP!, {R7,LR}
__text:000028C8 86 06 01 E3   MOV             R0, #0x1686
__text:000028CC 0D 70 A0 E1   MOV             R7, SP
__text:000028D0 00 00 40 E3   MOVT            R0, #0
__text:000028D4 00 00 8F E0   ADD             R0, PC, R0
__text:000028D8 C3 05 00 EB   BL              _puts
__text:000028DC 00 00 A0 E3   MOV             R0, #0
__text:000028E0 80 80 BD E8   LDMFD           SP!, {R7,PC}

__cstring:00003F62 48 65 6C 6C+aHelloWorld_0  DCB "Hello world!",0
```

Instrukcje STMFD i LDMFD są już nam znane.

Instrukcja MOV zapisuje liczbę 0x1686 do rejestru R0 — jest to przesunięcie, wskazujące na łańcuch znaków „Hello world!”.

Rejestr R7 (wg standardu [*iOS ABI Function Call Guide*, (2010)][39]) przechowuje wskaźnik ramki stosu (frame pointer). Będzie to omówione później.

Instrukcja MOVT R0, #0 (MOVe Top) zapisuje 0 do starszych 16 bitów rejestru. Zwykła instrukcja MOV w trybie ARM może zapisywać tylko do młodszych 16 bitów rejestru, z uwagi na długość instrukcji.

Warto pamiętać, że w trybie ARM kody operacji (opcode) instrukcji są ograniczone do 32 bitów. Nie ma to oczywiście wpływu na przenoszenie wartości między rejestrami. Z tego powodu do zapisywania do starszych bitów (od 16 do 31, włącznie) istnieje dodatkowa instrukcja MOVT. Tutaj jej użycie jest zbędne, gdyż MOV R0, #0x1686 i tak by wyzerowała starszą część rejestru. Możliwe, że jest to niedociągnięcie kompilatora.

Instrukcja ADD R0, PC, R0 dodaje PC do R0 żeby wyliczyć adres bezwzględny łańcucha znaków „Hello world!”. Jak już wiemy, jest to „position-independent code”, dlatego taka korekta jest niezbędna.

Instrukcja BL wywołuje puts() zamiast printf().

LLVM zamienił wywołanie printf() na puts(). Działanie printf() z jednym argumentem jest prawie równoznaczna puts().

Prawie, gdyż obie funkcje zadziałają identycznie, jeśli łańcuch znaków nie będzie zawierał sekwencji opisujących format, zaczynających się od znaku %. Jeśli będzie, wtedy wyniki ich pracy będą różne. [40].

Dlaczego kompilator zamienił wywoływaną funkcję? Prawdopodobnie dlatego, że funkcja puts() jest szybsza [41]. Najwidoczniej dlatego, że puts() przekazuje znaki na stdout, nie porównując ich ze znakiem %.

Dalej jest już znana instrukcja MOV R0, #0, ustawiająca 0 jako wartość zwracaną.

Optymalizujący Xcode 4.6.3 (LLVM) (tryb Thumb-2)

Domyślnie Xcode 4.6.3 wygeneruje trybie Thumb-2 podobny kod:

Listing 1.28: Optymalizujący Xcode 4.6.3 (LLVM) (tryb Thumb-2)

```
__text:00002B6C                   _hello_world
__text:00002B6C 80 B5           PUSH          {R7,LR}
__text:00002B6E 41 F2 D8 30     MOVW          R0, #0x13D8
__text:00002B72 6F 46           MOV           R7, SP
__text:00002B74 C0 F2 00 00     MOVT.W        R0, #0
__text:00002B78 78 44           ADD           R0, PC
__text:00002B7A 01 F0 38 EA     BLX           _puts
__text:00002B7E 00 20           MOVS          R0, #0
__text:00002B80 80 BD           POP           {R7,PC}
```

[39] Dostęp także przez http://developer.apple.com/library/ios/documentation/Xcode/Conceptual/iPhoneOSABIReference/iPhoneOSABIReference.pdf

[40] Należy również zauważyć, że puts() nie potrzebuje znaku nowej linii '\n' na końcu łańcucha, dlatego został on pominięty.

[41] ciselant.de/projects/gcc_printf/gcc_printf.html

```
...

__cstring:00003E70 48 65 6C 6C 6F 20+aHelloWorld  DCB "Hello world!",0xA,0
```

Instrukcje BL i BLX w trybie Thumb są kodowane jako para 16-bitowych instrukcji. W Thumb-2 te zastępcze kody operacji (opcode) rozszerzono tak, by instrukcje mogły być zakodowane jako 32-bitowe.

Można to łatwo zauważyc, gdyż w w trybie Thumb-2 wszystkie 32-bitowe instrukcje zaczynają się od `0xFx` lub `0xEx`.

Jednak na listingu w programie IDA bajty kodu operacji (opcode) są zamienione miejscami. W procesorze ARM instrukcje są kodowane w następujący sposób: najpierw podaje się ostatni bajt, potem pierwszy (dla trybów Thumb i Thumb-2), lub, dla trybu ARM, najpierw czwarty bajt, następnie trzeci, drugi i pierwszy (z uwagi na różną kolejność bajtów).

Bajty są wypisywane w listingach IDA w następującej kolejności:

- dla trybów ARM i ARM64: 4-3-2-1;
- dla trybu Thumb: 2-1;
- dla pary 16-bitowych instrukcji w trybie Thumb-2: 2-1-4-3.

Widzimy, że instrukcje `MOVW`, `MOVT.W` i `BLX` rzeczywiście zaczynają się od `0xFx`.

Jedną z tych instrukcji jest `MOVW R0, #0x13D8` — zapisuje ona 16-bitową liczbę do młodszych bitów rejestru `R0`, zerując starsze.

Inną instrukcją jest `MOVT.W R0, #0` — działa tak jak `MOVT` z poprzedniego przykładu, ale przeznaczona jest dla trybu Thumb-2

W tym przykładzie wykorzystana została instrukcja BLX zamiast BL. Różnica polega na tym, że oprócz zapisania adresu powrotu (RA) do rejestru LR i przekazania sterowania do funkcji `puts()`, odbywa się zmiana trybu procesora z Thumb/Thumb-2 na tryb ARM (lub odwrotnie).

Jest to niezbędne dlatego, że instrukcja, do której zostanie przekazane sterowanie jest zakodowana w trybie ARM i wygląda następująco:

```
__symbolstub1:00003FEC _puts           ; CODE XREF: _hello_world+E
__symbolstub1:00003FEC 44 F0 9F E5     LDR  PC, =__imp__puts
```

Jest to skok do miejsca, w którym, w sekcji importów, zapisany jest adres funkcji `puts()`. Można zadać pytanie: dlaczego nie można wywołać `puts()` bezpośrednio, tam gdzie jest to potrzebne?

Nie jest to efektywne z punktu widzenia oszczędności miejsca.

Praktycznie każdy program korzysta z zewnętrznych bibliotek łączonych dynamicznie (jak DLL w Windows, .so w *NIX czy .dylib w Mac OS X). W bibliotekach dynamicznych znajdują się często wykorzystywane funkcje biblioteczne, w tym funkcja `puts()` ze standardu C.

W wykonywalnym pliku binarnym (Windows PE .exe, ELF lub Mach-O) istnieje sekcja importów. Jest to lista symboli (funkcji lub zmiennych globalnych) importowanych z modułów zewnętrznych wraz z nazwami tych modułów. Program ładujący OS, iterując po symbolach zaimportowanych w module głównym, ładuje niezbędne moduły i ustawia rzeczywiste adresy każdego z symboli.

W naszym przypadku, *__imp__puts* jest 32-bitową zmienną, w której program ładujący OS umieści rzeczywisty adres funkcji z biblioteki zewnętrznej.

Następnie LDR odczytuje 32-bitową wartość z tej zmiennej, i zapisując ją do rejestru PC, przekazuje tam sterowanie.

Żeby skrócic czas tej procedury programu ładującego, trzeba sprawić aby adres każdego symbolu zapisywał się tylko raz, do specjalnie przydzielonego miejsca.

Do tego, jak się już upewniliśmy, zapisywanie 32-bitowej liczby do rejestru jest niemożliwe bez odwoływania się do pamięci.

Optymalnym rozwiązaniem jest wydzielenie osobnej funkcji, pracującej w trybie ARM, której jedynym celem jest przekazywanie sterowania dalej, do biblioteki dynamicznie łączonej. Nastepnie można wywoływać tę jednoinstrukcyjną funkcję z kodu w trybie Thumb.

Nawiasem mówiąc, w poprzednim przykładzie (skompilowanym dla trybu ARM), instrukcja BL przekazuje sterowanie do takiej samej thunk-funkcji, lecz procesor nie przestawia się w inny tryb (stąd brak „X” w mnemoniku instrukcji).

Jeszcze o thunk-funkcjach

Thunk-funkcje są trudne do zrozumienia przede wszystkim przez brak spójności w terminologii. Najprościej jest myśleć o nich jak o adapterach-przejściówkach z jednego typu gniazdek na drugi. Na przykład, adapter pozwalający włożyć do gniazdka amerykańskiego wtyczkę brytyjską lub na odwrót. Thunk-funkcje również są czasami nazywane *wrapper-ami*. *Wrap* w języku angielskim to *owinąć*, *zawinąć*, *opakować*. Oto jeszcze kilka definicji tych funkcji:

> “Kawałek kodu, który dostarcza adres:”, według P. Z. Ingerman, który wymyślił *thunk* w 1961 roku, jako sposób na powiązanie parametrów rzeczywistych z ich formalnymi definicjami w wywołaniach procedur, w języku Algol-60. Jeśli procedura jest wywołana z wyrażeniem w miejscu parametru formalnego, kompilator generuje *thunk*, który oblicza wartość wyrażenia i pozostawia adres tego wyniku w pewnej standardowej lokalizacji.
>
> …
>
> Microsoft i IBM zdefiniowali w ich systemach, opartych na Intelu, “środowisko 16-bitowe“ (z odrażającymi rejestrami segmentowymi i 64k limitem adresów) i “środowisko 32-bitowe“ (z płaskim adresowaniem i półrzeczywistym trybem zarządzania pamięcią). Te dwa środowiska mogą działać równocześnie na tym samym komputerze i systemie operacyjnym (dzięki temu, co w świecie Microsoftu znane jest jako

> WOW, co jest skrótowcem od Windows On Windows). MS i IBM zdecydowali, że proces przechodzenia z trybu 16-bitowego do 32-bitowego (i odwrotnie), nazwany zostanie “thunk”; istnieje nawet narzędzie na system Windows 95, ”THUNK.EXE”, nazwane ”thunk kompilatorem”.

(The Jargon File)

Jeszcze jeden przykład możemy znaleźć w bibliotece LAPACK — (“Linear Algebra PACKage”) napisanej w języku FORTRAN. Deweloperzy C/C++ również chcą korzystać z LAPACK, ale przepisywanie jej na C/C++, a następnie utrzymywanie kilku wersji byłoby szaleństwem. Istnieją wobec tego krótkie funkcje w C, które są wywoływane ze środowiska C/C++, które z kolei wywołują funkcje FORTRAN i prawie nic oprócz tego nie robią:

```
double Blas_Dot_Prod(const LaVectorDouble &dx, const LaVectorDouble &dy)
{
    assert(dx.size()==dy.size());
    integer n = dx.size();
    integer incx = dx.inc(), incy = dy.inc();

    return F77NAME(ddot)(&n, &dx(0), &incx, &dy(0), &incy);
}
```

Takie funkcje również są nazywane ”wrapperami”.

ARM64

GCC

Skompilujmy przykład w GCC 4.8.1 dla ARM64:

Listing 1.29: Nieoptymalizujący GCC 4.8.1 + objdump

```
0000000000400590 <main>:
  400590:       a9bf7bfd        stp     x29, x30, [sp,#-16]!
  400594:       910003fd        mov     x29, sp
  400598:       90000000        adrp    x0, 400000 <_init-0x3b8>
  40059c:       91192000        add     x0, x0, #0x648
  4005a0:       97ffffa0        bl      400420 <puts@plt>
  4005a4:       52800000        mov     w0, #0x0  // #0
  4005a8:       a8c17bfd        ldp     x29, x30, [sp],#16
  4005ac:       d65f03c0        ret

...

Contents of section .rodata:
 400640 01000200 00000000 48656c6c 6f210a00  ........Hello!..
```

W ARM64 nie ma trybów Thumb i Thumb-2, jest tylko ARM, więc wszystkie instrukcje są 32-bitowe.

Dysponujemy dwukrotnie większą liczbą rejestrów: **??** on page ??. 64-bitowe rejestry mają prefiks X-, a ich 32-bitowe części — W-.

Instrukcja STP (*Store Pair*) odkłada na stos jednocześnie 2 rejestry: X29 i X30. Oczywiście może ona zapisać tę parę gdziekolwiek w pamięci, ale w tym przypadku miejscem docelowym jest rejestr SP, a więc jest ona odkładana na stos.

Rejestry w ARM64 są 64-bitowe (8 bajtowe), dlatego do przechowywania 2 rejestrów potrzeba 16 bajtów.

Wykrzyknik ("!") po operandzie oznacza, że najpierw od SP będzie odjęte 16 i dopiero po tej czynności wartości z obu rejestrów będą odłożone na stos.

Jest to tak zwany *pre-index*. Więcej o różnicy między *post-index* a *pre-index*: **??** on page ??.

Posługując się terminologią x86 — pierwsza instrukcja jest analogiczna do pary instrukcji PUSH X29 i PUSH X30. X29 w ARM64 jest wykorzystywane jako FP[42], a X30 jako LR, dlatego są one odkładane na stos w prologu funkcji.

Druga instrukcja kopiuje SP do X29 (FP). Jest to niezbędne do ustawienia ramki stosu (stack frame) funkcji.

Instrukcje ADRP i ADD ustawiają adres łańcucha znaków „Hello!" w rejestrze X0, ponieważ pierwszy argument funkcji jest przekazywany przez ten rejestr. Jednakże w ARM nie ma instrukcji, za pomocą których można zapisać do rejestru dużą liczbę (dlatego że długość instrukcji wynosi maksymalnie 4 bajty. Więcej informacji o tym można znaleźć tutaj: **??** on page ??). Dlatego trzeba skorzystać z kilku instrukcji. Pierwsza instrukcja (ADRP) zapisuje do X0 adres strony o rozmiarze 4KiB, która zawiera łańcuch znaków, a druga (ADD) dodaje do tego adresu resztę (przesunięcie względem początku strony pamięci). Więcej o tym: **??** on page ??.

0x400000 + 0x648 = 0x400648, i możemy zobaczyć, że w segmencie danych .rodata pod tym adresem znajduje się nasz łańcuch znaków „Hello!".

Następnie za pomocą instrukcji BL jest wywoływana funkcja puts(). Zostało to omówione wcześniej: 1.5.3 on page 29.

Instrukcja MOV zapisuje 0 do W0. W0 to młodsze 32 bity 64-bitowego rejestru X0:

Starsze 32 bity	Młodsze 32 bity
X0	
	W0

Wynik funkcji jest zwracany przez X0 i main() zwraca 0.

Dlaczego 32-bitowa część? W ARM64, jak i w x86-64, typ *int* ma rozmiar 32 bitów, dla kompatybilności.

Skoro funkcja zwraca 32-bitowy *int*, to trzeba wypełnić tylko młodsze 32 bity 64-bitowego rejestru X0.

Żeby mieć pewność, zmieńmy przykład i skompilujmy go ponownie. Teraz main() zwraca 64-bitową wartość:

[42]Frame Pointer

Listing 1.30: funkcja `main()` zwracająca wartość typu `uint64_t`

```
#include <stdio.h>
#include <stdint.h>

uint64_t main()
{
        printf ("Hello!\n");
        return 0;
}
```

Wynik jest taki sam, tylko MOV w tej linii wygląda teraz:

Listing 1.31: Nieoptymalizujący GCC 4.8.1 + objdump

```
  4005a4:       d2800000        mov     x0, #0x0      // #0
```

Następnie za pomocą instrukcji LDP (*Load Pair*) przywracane są rejestry X29 i X30.

Nie ma wykrzyknika po instrukcji: oznacza to, że najpierw wartości są zdejmowane ze stosu a dopiero po tej czynności SP jest zwiększany o 16.

Jest to tzw. *post-index*.

W ARM64 pojawia się nowa instrukcja: RET. Działa tak samo jak BX LR, ale zawiera specjalny bit (ang. *hint bit*), który podpowiada procesorowi, że jest to wyjście z funkcji, a nie kolejna instrukcja skoku - by procesor mógł zoptymalizować jej wykonanie.

Funkcja jest bardzo prosta, GCC z włączoną optymalizacją generuje dokładnie taki sam kod.

1.5.4 MIPS

O „wskaźniku globalnym” („global pointer”)

„Wskaźnik globalny” („global pointer”) jest bardzo ważną koncepcją MIPS. Jak już wiemy, każda instrukcja w MIPS ma długość 32 bitów, dlatego niemożliwe jest zakodowanie 32-bitowego adresu w jednej instrukcji. Zamiast tego trzeba wykorzystać parę instrukcji (jak to robiło GCC dla załadowania adresu łańcucha znaków).

Można załadować dane z dowolnego adresu w przedziale $register - 32768...register + 32767$, za pomocą jednej instrukcji, dlatego że można w niej zakodować 16-bitowe przesunięcie (przesunięcie może być ujemne, stąd mówimy o zakresie liczby 16-bitowej ze znakiem). Możemy więc przydzielić do tego celu jakiś rejestr i zaalokować bufor 64KiB na najczęściej wykorzystywane dane.

Rejestr ten nazywamy „wskaźnikiem globalnym” („global pointer”) i wskazuje on na środek bufora 64KiB. Bufor zwykle zawiera zmienne globalne oraz adresy funkcji importowanych, jak `printf()`, ponieważ deweloperzy GCC stwierdzili, że pobranie adresu pewnych funkcji musi być tak szybkie, jak to możliwe, i powinno zająć jedną instrukcję, a nie dwie.

W plikach ELF ten 64KiB bufor znajduje się częściowo w sekcji .sbss („small BSS[43]”)

[43]Block Started by Symbol

dla danych niezainicjalizowanych i .sdata („small data”) dla danych zainicjalizowanych. To oznacza, że programista może wybrać do czego potrzebuje szybszego dostępu i umieścić to w segmentach .sdata/.sbss. Niektórzy programiści starej daty mogą pamiętać model pamięci w MS-DOS **??** on page ?? lub w managery pamięci typu XMS/EMS, gdzie cała pamięć była podzielona na bloki o długości 64KiB.

Ten pomysł jest wykorzystywany również w innych architekturach, np. PowerPC.

Optymalizujący GCC

Popatrzmy na poniższy przykład, pokazujący wykorzystanie „wskaźnika globalnego”.

Listing 1.32: Optymalizujący GCC 4.4.5 (wyjście w asemblerze)

```
$LC0:
; \000 to bajt zero w systemie ósemkowym:
        .ascii  "Hello, world!\012\000"
main:
; prolog funkcji
; ustaw GP:
        lui     $28,%hi(__gnu_local_gp)
        addiu   $sp,$sp,-32
        addiu   $28,$28,%lo(__gnu_local_gp)
; odłóż RA na stos lokalny:
        sw      $31,28($sp)
; załaduj adres funkcji puts() z GP do $25:
        lw      $25,%call16(puts)($28)
; załaduj adres łańcucha znaków do $4 ($a0):
        lui     $4,%hi($LC0)
; skocz do puts(), zapisując adres powrotu do rejestru powrotu:
        jalr    $25
        addiu   $4,$4,%lo($LC0) ; branch delay slot
; przywróć RA:
        lw      $31,28($sp)
; skopiuj 0 z $zero do $v0:
        move    $2,$0
; zwróć sterowanie, skacząc pod adres w RA:
        j       $31
; epilog funkcji:
        addiu   $sp,$sp,32 ; branch delay slot + posprzątaj stos lokalny
```

Rejestr $GP w prologu funkcji ustawiany jest na środek tego obszaru. Na stos lokalny odkładany jest rejestr RA. W tym przykładzie kompilator również zamienił wywołanie `printf()` na `puts()`. Adres funkcji `puts()` jest ładowany do $25 za pomocą instrukcji `LW` („Load Word”). Następnie adres łańucha znaków jest ładowany do $4 za pomocą pary instrukcji `LUI` („Load Upper Immediate”) i `ADDIU` („Add Immediate Unsigned Word”). `LUI` ustawia starsze 16 bitów rejestru (stąd „upper” w nazwie instrukcji) i `ADDIU` dodaje młodsze 16 bitów do adresu.

`ADDIU` znajduje się po `JALR` (pamiętaj jednak o *branch delay slot*). Rejestr $4 ($A0) jest wykorzystywany do przekazywania pierwszego argumentu funkcji [44].

[44] Tabela rejestrów MIPS znajduje się w dodatku **??** on page ??

JALR („Jump and Link Register”) skacze pod adres z rejestru $25 (znajduje się w nim adres `puts()`), jednocześnie zapisując adres instrukcji, która zostanie wywołana jako następna (LW) w RA. Widać podobieństwo do ARM. I jeszcze jedna bardzo ważna rzecz: adres zapisywany do RA nie jest adresem kolejnej (ADDIU) instrukcji z listingu (dlatego, że jest to *delay slot* i wykonuje się przed instrukcją skoku), a instrukcji następującej po *delay slot*. W ten sposób, podczas wykonywania JALR, do RA jest zapisywane $PC + 8$. W naszym wypadku jest to adres instrukcji LW, kolejnej po ADDIU.

LW („Load Word”) w linii 20 przywraca RA ze stosu lokalnego (ta instrukcja jest właściwie częścią epilogu funkcji).

MOVE w linii 22 kopiuje wartość z $0 ($ZERO) do $2 ($V0).

MIPS ma specjalny rejestr, który zawsze zawiera stałą zero. Najwyraźniej deweloperzy MIPS stwierdzili, że 0 jest najpowszechniejszą stałą w programowaniu, więc niech rejestr $0 będzie wykorzystywany za każdym razem, kiedy będzie potrzebne 0.

Inna ciekawostka: w MIPS nie ma instrukcji kopiującej wartość z rejestru do rejestru. `MOVE DST, SRC` to w rzeczywistości `ADD DST, SRC, $ZERO` (gdzie $DST = SRC + 0$). Najwidoczniej twórcy MIPS chcieli stworzyć jak najbardziej zwięzłą tablicę kodów operacji (opcode). To wcale nie znaczy, że dodawanie jest wykonywane podczas każdej instrukcji MOVE. Prawdopodobnie te pseudoinstrukcje są optymalizowane w CPU i ALU[45] nigdy nie jest wykorzystywane.

J w linii 24 skacze pod adres w RA, co powoduje wyjście z funkcji. ADDIU poniżej J jest wykonywane przed J (pamiętasz o *branch delay slot*?) i należy do epilogu funkcji.

Poniżej listing z programu IDA. Każdy rejestr posiada swoją pseudonazwę:

Listing 1.33: Optymalizujący GCC 4.4.5 (IDA)

```
.text:00000000 main:
.text:00000000
.text:00000000 var_10          = -0x10
.text:00000000 var_4           = -4
.text:00000000
; prolog funkcji
; ustaw GP:
.text:00000000                 lui     $gp, (__gnu_local_gp >> 16)
.text:00000004                 addiu   $sp, -0x20
.text:00000008                 la      $gp, (__gnu_local_gp & 0xFFFF)
; odłóż RA na stos lokalny:
.text:0000000C                 sw      $ra, 0x20+var_4($sp)
; odłóż GP na stos lokalny:
; z jakiegoś powodu tej instrukcji nie było na listingu z GCC:
.text:00000010                 sw      $gp, 0x20+var_10($sp)
; załaduj adres funkcji puts() z GP do $t9:
.text:00000014                 lw      $t9, (puts & 0xFFFF)($gp)
; wylicz adres łańcucha znaków i zapisz w $a0:
.text:00000018                 lui     $a0, ($LC0 >> 16)  # "Hello, world!"
; skocz do puts(), zapisując adres powrotu do rejestru powrotu
.text:0000001C                 jalr    $t9
.text:00000020                 la      $a0, ($LC0 & 0xFFFF)  # "Hello,
    world!"
```

[45] Jednostka arytmetyczno-logiczna (Arithmetic Logic Unit)

```
23 ; przywróć RA:
24 .text:00000024                  lw      $ra, 0x20+var_4($sp)
25 ; skopiuj 0 z $zero do $v0:
26 .text:00000028                  move    $v0, $zero
27 ; zwróć sterowanie, skacząc pod adres z RA:
28 .text:0000002C                  jr      $ra
29 ; epilog funkcji:
30 .text:00000030                  addiu   $sp, 0x20
```

Instrukcja w linii 15 odkłada GP na stos lokalny. Co ciekawe, brakuje jej na listingu z GCC — możliwe, że jest to spowodowane błędem w samym GCC[46]. Wartość GP musi zostać odłożona, ponieważ każda funkcja może używać swojego własnego 64KiB bufora. Rejestr zawierający adres `puts()` nazwany został $T9, dlatego że rejestry z prefiksem T określane są jako „tymczasowe” i ich zawartości nie musi być zachowywana - nie jest więc odkładana na stos.

Nieoptymalizujący GCC

Nieoptymalizujący GCC generuje nieco rozwlekły kod.

Listing 1.34: Nieoptymalizujący GCC 4.4.5 (wyjście w asemblerze)

```
$LC0:
        .ascii  "Hello, world!\012\000"
main:
; prolog funkcji
; odłóż RA ($31) i FP na stos:
        addiu   $sp,$sp,-32
        sw      $31,28($sp)
        sw      $fp,24($sp)
; ustaw FP (stack frame pointer):
        move    $fp,$sp
; ustaw GP:
        lui     $28,%hi(__gnu_local_gp)
        addiu   $28,$28,%lo(__gnu_local_gp)
; załaduj adres łańcucha znaków
        lui     $2,%hi($LC0)
        addiu   $4,$2,%lo($LC0)
; załaduj adres funkcji puts() z GP:
        lw      $2,%call16(puts)($28)
        nop
; wywołaj puts():
        move    $25,$2
        jalr    $25
        nop  ; branch delay slot

; przywróć GP ze stosu lokalnego:
        lw      $28,16($fp)
; ustaw rejestr $2 ($V0) na zero:
        move    $2,$0
```

[46] Najwidoczniej funkcje generujące listingi nie są krytyczne dla użytkowników GCC, dlatego pewne kosmetyczne błędy wciąż mogą być niepoprawione.

```
; epilog funkcji.
; przywróć SP:
        move    $sp,$fp
; przywróć RA:
        lw      $31,28($sp)
; przywróć FP:
        lw      $fp,24($sp)
        addiu   $sp,$sp,32
; skok pod adres z RA:
        j       $31
        nop  ; branch delay slot
```

FP jest wykorzystywany jako wskaźnik ramki stosu (stack frame pointer). Widać również 3 instrukcje NOP. Drugi i trzeci NOP występują po instrukcjach skoku. Prawdopodobnie kompilator GCC zawsze dodaje NOP-y (przez *branch delay slot*) po instrukcjach skoku a następnie, jeśli optymalizacja jest włączona, może je wyeliminować. W tym przypadku instrukcje pozostały na swoim miejscu.

Poniżej ten sam plik wykonywalny w programie IDA:

Listing 1.35: Nieoptymalizujący GCC 4.4.5 (IDA)

```
.text:00000000 main:
.text:00000000
.text:00000000 var_10          = -0x10
.text:00000000 var_8           = -8
.text:00000000 var_4           = -4
.text:00000000
; prolog funkcji
; odłóż RA i FP na stos:
.text:00000000                 addiu   $sp, -0x20
.text:00000004                 sw      $ra, 0x20+var_4($sp)
.text:00000008                 sw      $fp, 0x20+var_8($sp)
; ustaw FP (stack frame pointer):
.text:0000000C                 move    $fp, $sp
; ustaw GP:
.text:00000010                 la      $gp, __gnu_local_gp
.text:00000018                 sw      $gp, 0x20+var_10($sp)
; załaduj adres łańcucha znaków:
.text:0000001C                 lui     $v0, (aHelloWorld >> 16)  # "Hello,
    world!"
.text:00000020                 addiu   $a0, $v0, (aHelloWorld & 0xFFFF)  #
    "Hello, world!"
; załaduj adres funkcji puts() z GP:
.text:00000024                 lw      $v0, (puts & 0xFFFF)($gp)
.text:00000028                 or      $at, $zero ; NOP
; wywołaj puts():
.text:0000002C                 move    $t9, $v0
.text:00000030                 jalr    $t9
.text:00000034                 or      $at, $zero ; NOP
; przywróć GP ze stosu lokalnego:
.text:00000038                 lw      $gp, 0x20+var_10($fp)
; ustaw rejestr $2 ($V0) na zero:
.text:0000003C                 move    $v0, $zero
; epilog funkcji.
```

```
32 ; przywróć SP:
33 .text:00000040                  move    $sp, $fp
34 ; przywróć RA:
35 .text:00000044                  lw      $ra, 0x20+var_4($sp)
36 ; przywróć FP:
37 .text:00000048                  lw      $fp, 0x20+var_8($sp)
38 .text:0000004C                  addiu   $sp, 0x20
39 ; skocz pod adres z RA:
40 .text:00000050                  jr      $ra
41 .text:00000054                  or      $at, $zero ; NOP
```

Co ciekawe, IDA rozpoznała parę LUI/ADDIU i zebrała ją w jedną pseudoinstrukcję LA („Load Address”) w linii 15. Można zauważyć, że jej długość to 8 bajtów! Nazywamy to pseudoinstrukcją (lub *makrem*), ponieważ nie jest to prawdziwa instrukcja MIPS, tylko wygodna nazwa dla pary dwóch powiązanych instrukcji.

Widać również, że IDA nie rozpoznała instrukcji NOP w liniach 22, 26 i 41.

W rzeczywistości NOP jest realizowany przez `OR $AT, $ZERO`. Jest to instrukcja przeprowadzająca operację *LUB* na rejestrze $AT i $ZERO (rejestr zawsze przechowujący stałą 0), co jest pustą instrukcją. MIPS, jak i niektóre inne ISA, nie posiada oddzielnej instrukcji NOP.

Rola ramki stosu (stack frame) w tym przykładzie

Adres łańcucha znaków jest przekazywany przez rejestr. Po co w takim razie ustawiać stos lokalny? Wartości rejestrów RA i GP muszą być gdzieś zapisane (ponieważ wywołujemy funkcję - `printf()`) i w tym celu korzysta się ze stosu lokalnego.

Gdyby to była funkcja liść (ang. leaf function), to można by pozbyć się prologu i epilogu funkcji, zobacz przykład: 1.4.3 on page 11.

Optymalizujący GCC: sesja w debuggerze GDB

Listing 1.36: Przykład sesji w GDB

```
root@debian-mips:~# gcc hw.c -O3 -o hw

root@debian-mips:~# gdb hw
GNU gdb (GDB) 7.0.1-debian
...
Reading symbols from /root/hw...(no debugging symbols found)...done.
(gdb) b main
Breakpoint 1 at 0x400654
(gdb) run
Starting program: /root/hw

Breakpoint 1, 0x00400654 in main ()
(gdb) set step-mode on
(gdb) disas
Dump of assembler code for function main:
0x00400640 <main+0>:    lui     gp,0x42
0x00400644 <main+4>:    addiu   sp,sp,-32
```

```
0x00400648 <main+8>:    addiu   gp,gp,-30624
0x0040064c <main+12>:   sw      ra,28(sp)
0x00400650 <main+16>:   sw      gp,16(sp)
0x00400654 <main+20>:   lw      t9,-32716(gp)
0x00400658 <main+24>:   lui     a0,0x40
0x0040065c <main+28>:   jalr    t9
0x00400660 <main+32>:   addiu   a0,a0,2080
0x00400664 <main+36>:   lw      ra,28(sp)
0x00400668 <main+40>:   move    v0,zero
0x0040066c <main+44>:   jr      ra
0x00400670 <main+48>:   addiu   sp,sp,32
End of assembler dump.
(gdb) s
0x00400658 in main ()
(gdb) s
0x0040065c in main ()
(gdb) s
0x2ab2de60 in printf () from /lib/libc.so.6
(gdb) x/s $a0
0x400820:        "hello, world"
(gdb)
```

1.5.5 Wnioski

Główną różnicą między kodem w x86/ARM a x64/ARM64 jest to, że wskaźnik na łańcuch znaków stał się 64-bitowy. W rzeczy samej, współczesne CPU stały się 64-bitowe przez niższy koszt pamięci oraz większe zapotrzebowania na nią przez współczesne aplikacje. Komputery mogą mieć więcej pamięci, niż można zaadresować za pomocą 32 bitów. W związku z tym, wszystkie wskaźniki są teraz 64-bitowe.

1.5.6 Ćwiczenia

- http://challenges.re/48
- http://challenges.re/49

1.6 Prolog i epilog funkcji

Prolog funkcji to sekwencja instrukcji rozpoczynająca funkcję. Zwykle przypomina poniższy fragment kodu:

```
    push    ebp
    mov     ebp, esp
    sub     esp, X
```

Co te instrukcje robią: zapisują wartość rejestru EBP na stosie, ustawiają wartość w rejestrze EBP na wartość z ESP a następnie alokuję miejsce na stosie na zmienne lokalne

Wartość w EBP pozostaje niezmieniona przez całą funkcję i używana jest przy dostępie do zmiennych lokalnych i argumentów. Do tego samego celu można by użyć ESP, ale byłoby to niewygodne, gdyż wartość ESP zmienia się w czasie wykonywania funkcji.

Epilog funkcji zwalnia zaalokowane miejsce na stosie, przywraca wartość rejestru EBP do jego pierwotnego stanu i zwraca sterowanie do funkcji wywołującej:

```
    mov     esp, ebp
    pop     ebp
    ret     0
```

Prolog i epilog zwykle jest wykrywany przez deasemblery i używany do wyodrębnienia pojedynczych funkcji.

1.6.1 Rekurencja

Prolog i epilog funkcji mają negatywny wpływ na wywołania rekurencyjne.

Więcej o rekurencji: **??** on page ??.

1.7 Pusta funkcja raz jeszcze

Wróćmy do przykładu z pustą funkcją 1.3 on page 8. Uzbrojeni w wiedzę o prologu i epilogu funkcji, spójrzmy na wynik kompilacji GCC bez optymalizacji:

Listing 1.37: Nieoptymalizujący GCC 8.2 x64 (wyjście w asemblerze)

```
f:
        push    rbp
        mov     rbp, rsp
        nop
        pop     rbp
        ret
```

Poza instrukcją RET jest tam jedynie prolog i epilog, które nie zostały zoptymalizowane.

NOP wygląda jak kolejna osobliwość kompilatora. Jedyną instrukcją, która realizuje działanie programu, jest tutaj RET. Wszystkie pozostałe można by usunąć (zoptymalizować).

1.8 Zwracanie wartości raz jeszcze

Znając pojęcie prologu i epilogu, skompilujmy raz jeszcze przykład z wartością zwracaną (1.4 on page 10, 1.8 on page 10) za pomocą GCC z wyłączoną optymalizacją:

Listing 1.38: Nieoptymalizujący GCC 8.2 x64 (wyjście w asemblerze)

```
f:
        push    rbp
```

```
        mov     rbp, rsp
        mov     eax, 123
        pop     rbp
        ret
```

Instrukcje realizujące działanie programu to MOV i RET – pozostałe to prolog i epilog.

1.9 Stos

Stos w informatyce jest jedną z najbardziej fundamentalnych struktur danych [47]. AKA[48] LIFO[49].

Technicznie rzecz biorąc, jest to blok pamięci w pamięci procesu + rejestr ESP w x86, RSP w x64, lub SP w ARM, który przechowuje wskaźnik na miejsce w granicach tego bloku.

Najczęściej używanymi instrukcjami do operowania na stosie są PUSH i POP (w x86 i trybie Thumb w ARM). PUSH zmniejsza ESP/RSP/SP o 4 w trybie 32-bitowym (lub o 8 w 64-bitowym), następnie zapisuje zawartość swojego operandu pod adres, na który wskazuje ESP/RSP/SP, .

POP jest odwrotną operacją: najpierw pobiera dane z miejsca, na które wskazuje wskaźnik stos i umieszcza ją w miejscu wskazywanym przez operand docelowy (często jest to rejestr), a następnie zwiększa wskaźnik stosu o 4 (lub 8).

Po zaalokowaniu stosu wskaźnik stosu pokazuje na koniec tego obszaru pamięci. PUSH zmniejsza wskaźnik stosu, a POP — zwiększa. Koniec stosu znajduje się na początku zaalokowanego bloku pamięci. Może zabrzmieć to dziwnie, ale tak to działa.

ARM wspiera zarówno stosy rosnący w dół, jak i w górę.

Na przykład instrukcje STMFD/LDMFD, STMED[50]/LDMED[51] są przeznaczone dla stosu malejącego (rosnącego w dół od adresów wysokich do adresów niskich).
Natomiast instrukcje STMFA[52]/LDMFA[53], STMEA[54]/LDMEA[55] są przeznaczone dla stosu rosnącego (rosnącego w górę od niskich adresów do adresów wysokich).

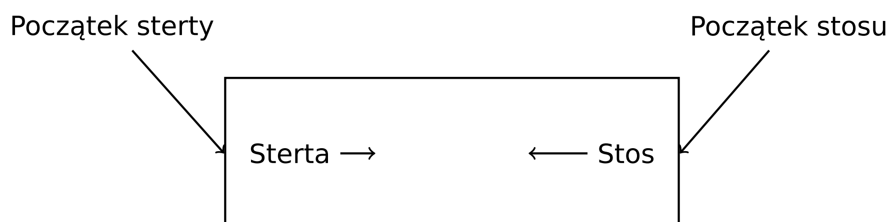
1.9.1 Dlaczego stos rośnie w dół?

Intuicyjnie moglibyśmy pomyśleć, że jak każda inna struktura danych, stos mógłby rosnąć w górę, w kierunku adresów wysokich.

Prawdopodobnie stos rośnie w dół ze względów historycznych. Kiedy komputery były duże i zajmowały cały pokój, można było bardzo łatwo podzielić pamięć na dwa

[47] wikipedia.org/wiki/Call_stack
[48] Also Known As — znany również jako
[49] Ostatni na wejściu, pierwszy na wyjściu (Last In First Out)
[50] Store Multiple Empty Descending ()
[51] Load Multiple Empty Descending ()
[52] Store Multiple Full Ascending ()
[53] Load Multiple Full Ascending ()
[54] Store Multiple Empty Ascending ()
[55] Load Multiple Empty Ascending ()

obszary: na stertę i na stos. Nie było wiadomo z góry, jak duża może być sterta lub stos, dlatego takie rozwiązanie było najprostsze.



W [D. M. Ritchie and K. Thompson, *The UNIX Time Sharing System*, (1974)][56]można przeczytać:

> Dane użytkownika (część obrazu procesu) podzielone są na trzy logiczne segmenty. Segment kodu programu zaczyna się od adresu 0 w wirtualnej przestrzeni adresowej. W trakcie wykonania jest on zabezpieczony przed zapisem i tylko jedna jego kopia jest współdzielona przez wszystkie procesy, wykonujące ten sam program. Po pierwszej granicy 8KB ponad segmentem kodu programy, w wirtualnej przestrzeni adresowej, rozpoczyna się prywatny, zapisywalny segment danych, którego rozmiar może być zwiększany przez wywołanie systemowe. Od najwyższego adresu w wirtualnej przestrzeni adresowej zaczyna się segment stosu, który automatycznie rośnie w dół stosownie do zmian wskaźnika stosu.

To trochę przypomina sytuację, gdy uczeń prowadzi notatki z dwóch wykładów w jednym zeszycie. Pierwsze notatki zaczynają się konwencjonalnie, od początku zeszytu, ale drugie zapisywane są na końcu, po obróceniu zeszytu. Gdy zabraknie miejsca, notatki spotkają się gdzieś w środku.

1.9.2 Do czego wykorzystywany jest stos?

Zapisywanie adresu powrotu

x86

Przed wywołaniem funkcji za pomocą instrukcji `CALL`, na stos odkładany jest adres kolejnej instrukcji (tej bezpośrednio za `CALL`). Następnie następuje skok bezwarunkowy pod adres z operandu instrukcji `CALL`.

Instrukcja `CALL` jest równoważna parze instrukcji `PUSH adres_docelowy / JMP`.

RET zdejmuje adres ze stosu i przekazuje tam sterowanie — jest to równoważne parze instrukcji `POP tmp / JMP tmp`.

Bardzo łatwo przepełnić stos, poprzez nieskończoną rekurencję:

```
void f()
```

[56]Dostęp także przez URL

```
{
        f();
};
```

MSVC 2008 wyświetli ostrzeżenie:

```
c:\tmp6>cl ss.cpp /Fass.asm
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 15.00.21022.08 for ↙
    ↘ 80x86
Copyright (C) Microsoft Corporation.  All rights reserved.

ss.cpp
c:\tmp6\ss.cpp(4) : warning C4717: 'f' : recursive on all control paths, ↙
    ↘ function will cause runtime stack overflow
```

...ale wygeneruje plik wykonywalny:

```
?f@@YAXXZ PROC                          ; f
; Line 2
        push    ebp
        mov     ebp, esp
; Line 3
        call    ?f@@YAXXZ               ; f
; Line 4
        pop     ebp
        ret     0
?f@@YAXXZ ENDP                          ; f
```

...jeśli włączymy optymalizację (opcja /0x), to zoptymalizowany kod nie będzie powodował przepełnienia stosu i działał *poprawnie*[57]

```
?f@@YAXXZ PROC                          ; f
; Line 2
$LL3@f:
; Line 3
        jmp     SHORT $LL3@f
?f@@YAXXZ ENDP                          ; f
```

GCC 4.4.1 wygeneruje taki sam kod w obu przypadkach i nie wyświetli żadnego ostrzeżenia.

ARM

Programy na ARM również korzystają ze stosu do zapisywania adresu powrotu, ale w trochę inny sposób. Jak już było wspomniane w rozdziale „Hello, world!” (1.5.3 on page 25), adres powrotu (RA) jest zapisywany do rejestru LR (rejestr powrotu). Jeśli zajdzie potrzeba wywołania kolejnej funkcji i ponownego użycia LR, to jego zawartość będzie musiała być gdzieś zapisana.

[57] o ironio!

Zwykle odbywa się to w prologu funkcji, często widzimy tam instrukcję jak `PUSH {R4-R7,LR}`, a w epilogu `POP {R4-R7,PC}` — rejestry, z których będzie korzystała bieżąca funkcja, w tym rejestr LR, odkładane są na stos.

Jeśli jakaś funkcja nie wywołuje żadnych innych funkcji w trakcie swojej pracy, w terminologii RISC nazywana jest *funkcją-liściem*[58]. Z tego powodu funkcja-liść nie odkłada rejestru LR na stos, ponieważ go nie zmienia. Jeśli funkcja jest niewielkich rozmiarów i korzysta z małej liczby rejestrów, to może w ogóle nie korzystać ze stosu. Stąd w ARM możliwe jest wywoływanie małych funkcji-liści bez używania stosu. Jest to szybsze niż w starych x86, gdyż nie korzysta się z pamięci zewnętrznej RAM do korzystania ze stosu. [59]. Ten mechanizm przydaje się również, gdy pamięć pod stos nie została jeszcze zaalokowana albo jest niedostępna.

kilka przykładów takich funkcji: 1.14.3 on page 137, 1.14.3 on page 138, **??** on page ??, **??** on page ??, **??** on page ??, **??** on page ??, **??** on page ??, **??** on page ??.

Przekazywanie argumentów funkcji

Najpopularniejszy sposób na przekazywanie parametrów funkcji w x86 to „cdecl”:

```
push arg3
push arg2
push arg1
call f
add esp, 12 ; 4*3=12
```

Wywoływana funkcja pobiera swoje argumentu za pomocą wskaźnik stosu.

Stos, przed wykonaniem pierwszej instrukcji z `f()`, wygląda następująco:

ESP	adres powrotu
ESP+4	argument#1, oznaczony w programie IDA jako `arg_0`
ESP+8	argument#2, oznaczony w programie IDA jako `arg_4`
ESP+0xC	argument#3, oznaczony w programie IDA jako `arg_8`
…	…

Opis innych konwencji wywoływania funkcji znajduje się tutaj: (**??** on page ??).

Nawiasem mówiąc, funkcja wywoływana nie posiada informacji o liczbie przekazywanych do niej argumentów. Funkcja `printf()`, która może mieć zmienną liczbę argumentów, ustala ich liczbę za pomocą specyfikatorów formatu (rozpoczynających się od znaku %).

Jeśli napiszemy:

```
printf("%d %d %d", 1234);
```

`printf()` wypisze 1234, a następnie jeszcze dwie losowe[60] liczby, który przypadkowo znalazły się na stosie obok.

[58]infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.faqs/ka13785.html
[59]Kiedyś, na PDP-11 i VAX, instrukcja CALL (wywołanie innych funkcji) była kosztowna; procesor spędzał na CALL nawet do 50% czasu wykonania programu. Z tego powodu posiadanie dużej liczby małych funkcji uchodziło za antywzorzec [Eric S. Raymond, *The Art of UNIX Programming*, (2003)Chapter 4, Part II].
[60]Tak na prawdę nie są one losowe, patrz: 1.9.4 on page 52

Dlatego nie ma znaczenia jak zapiszemy funkcję `main()`:
jak `main()`, `main(int argc, char *argv[])`
lub `main(int argc, char *argv[], char *envp[])`.

W rzeczywistości kod z CRT wywołuje `main()` mniej więcej tak:

```
push envp
push argv
push argc
call main
...
```

Jeśli zadeklarujesz `main()` bez argumentów, one i tak będą na stosie, lecz nie zostaną wykorzystane. Jeśli zadeklarujesz `main()` jako `main(int argc, char *argv[])`, to będziesz mógł skorzystać z pierwszych dwóch argumentów, a trzeci będzie dla funkcji „niewidoczny”. Co więcej, można nawet zadeklarować `main(int argc)` i to również zadziała.

Inny, podobny, przykład: **??**.

Alternatywne sposoby na przekazywanie argumentów

Warto zauważyć, że nic nie zmusza programisty do przekazywanie argumentów przez stos. Nie ma takiego wymagania, można to robić to zupełnie inaczej, nie korzystając ze stosu.

Dość popularnym sposobem wśród początkujących jest przekazywanie argumentów przez zmienne globalne, na przykład:

Listing 1.39: Kod w asemblerze

```
        ...

        mov     X, 123
        mov     Y, 456
        call    do_something

        ...

X       dd      ?
Y       dd      ?

do_something proc near
        ; take X
        ; take Y
        ; do something
        retn
do_something endp
```

Ta metoda posiada oczywistą wadę: funkcja *do_something()* nie może wywołać sama siebie przez rekurencję (lub za pomocą innej funkcji), gdyż musiałaby nadpisać własne argumenty. To samo dotyczy zmiennych lokalnych, gdyby przechowywać je w

zmiennych globalnych, to funkcja nie będzie mogła wywołać sama siebie. Co więcej, nie jest to bezpieczne w środowisku wielowątkowym[61]. Przechowywania danych na stosie wszystko upraszcza — stos może przechować tyle argumentów funkcji/zmiennych, na ile pozwoli jego rozmiar.

W [Donald E. Knuth, *The Art of Computer Programming*, Volume 1, 3rd ed., (1997), 189] można przeczytać o jeszcze dziwniejszych metodach przekazywania argumentów, szczególnie wygodnych na IBM System/360.

W MS-DOS istniała metoda przekazywania argumentów przez rejestry, na przykład ten fragment kodu na wiekowym 16-bitowym MS-DOS wypisze „Hello, world!”:

```
mov  dx, msg      ; adres wiadomości
mov  ah, 9        ; 9 oznacza funkcję "wypisz łańcuch znaków"
int  21h          ; wywołanie systemowe ("syscall") DOS

mov  ah, 4ch      ; funkcja "zakończ program"
int  21h          ; wywołanie systemowe ("syscall") DOS

msg  db 'Hello, World!\$'
```

Jest to całkiem podobne do metody **??** on page ??. Przypomina to również sposoby korzystania z wywołań systemowych (ang. syscall) na systemach Linuks (**??** on page ??) i Windows.

Jeżeli funkcja w MS-DOS zwraca wartość typu boolean (jeden bit, zwykle oznaczający wystąpienie błędu), to często wykorzystywana jest flaga CF.

Na przykład:

```
mov ah, 3ch       ; "3c" oznacza "stwórz plik
lea dx, filename
mov cl, 1
int 21h
jc  error
mov file_handle, ax
...
error:
...
```

W razie wystąpienia błędu flaga CF zostaje ustawiona i instrukcja JC skacze do miejsca oznaczonego etykietą error. W przeciwnym razie uchwyt (ang. *file handle*) stworzonego pliku zwracany jest przez rejestr AX.

Ta metoda wciąż jest wykorzystywana przez programistów asemblera. W kodach źródłowych Windows Research Kernel (który jest bardzo podobny do Windows 2003) możemy znaleźć coś takiego
(plik *base/ntos/ke/i386/cpu.asm*):

```
        public  Get386Stepping
Get386Stepping  proc
```

[61] W poprawnej implementacji każdy wątek miałby własny stos lokalny, ze swoimi argumentami/zmiennymi.

```
        call    MultiplyTest                ; Perform multiplication test
        jnc     short G3s00                 ; if nc, muttest is ok
        mov     ax, 0
        ret
G3s00:
        call    Check386B0                  ; Check for B0 stepping
        jnc     short G3s05                 ; if nc, it's B1/later
        mov     ax, 100h                    ; It is B0/earlier stepping
        ret

G3s05:
        call    Check386D1                  ; Check for D1 stepping
        jc      short G3s10                 ; if c, it is NOT D1
        mov     ax, 301h                    ; It is D1/later stepping
        ret

G3s10:
        mov     ax, 101h                    ; assume it is B1 stepping
        ret

        ...

MultiplyTest    proc

        xor     cx,cx                       ; 64K times is a nice round number
mlt00:  push    cx
        call    Multiply                    ; does this chip's multiply work?
        pop     cx
        jc      short mltx                  ; if c, No, exit
        loop    mlt00                       ; if nc, YEs, loop to try again
        clc
mltx:
        ret

MultiplyTest    endp
```

Przechowywanie zmiennych lokalnych

Funkcja może zaalokować miejsce na stosie dla własnych zmiennych lokalnych przez zmniejszenie wskaźnika stosu, w kierunku końca stosu (pamiętaj, że stos rośnie w dół, w kierunku niskich adresów!).

Jest to bardzo szybkie, niezależnie od liczby zmiennych lokalnych. Wiedz, że nie ma przymusu trzymania zmiennych lokalnych na stosie. Możesz je trzymać gdziekolwiek, ale tradycyjnie wykorzystuje się do tego stos.

x86: Funkcja alloca()

Ciekawym przypadkiem jest funkcja `alloca()` [62]. Działa ona jak `malloc()`, ale przydziela pamięć bezpośrednio na stosie. Nie ma potrzeby zwalniania tak zaalokowanego obszaru pamięci za pomocą `free()`, gdyż epilog funkcji (1.6 on page 40) przywróci ESP do stanu początkowego i zaalokowana pamięć zostanie *porzucona*. Ciekawa jest również implementacja tej funkcji. Krótko mówiąc, przesuwa ESP w dół stosu o wymaganą liczbę bajtów, przez co ESP wskazuje na przydzielony obszar pamięci.

Sprawdźmy:

```
#ifdef __GNUC__
#include <alloca.h> // GCC
#else
#include <malloc.h> // MSVC
#endif
#include <stdio.h>

void f()
{
    char *buf=(char*)alloca (600);
#ifdef __GNUC__
    snprintf (buf, 600, "hi! %d, %d, %d\n", 1, 2, 3); // GCC
#else
    _snprintf (buf, 600, "hi! %d, %d, %d\n", 1, 2, 3); // MSVC
#endif

    puts (buf);
};
```

Funkcja `_snprintf()` działa tak samo jak `printf()`, tylko zamiast wypisywać tekst na standardowe wyjście (stdout), zapisuje do bufora `buf`. Z kolei funkcja `puts()` kopiuje zawartość bufora `buf` na standardowe wyjście. Oczywiście zamiast korzystać z tych dwóch funkcji, można by użyć `printf()`, ale chcemy zobaczyć wykorzystanie niewielkiego bufora.

MSVC

Skompilujmy (MSVC 2010):

Listing 1.40: MSVC 2010

```
...

    mov    eax, 600  ; 00000258H
    call   __alloca_probe_16
    mov    esi, esp

    push   3
    push   2
```

[62] W MSVC implementację funkcji można podejrzeć w plikach `alloca16.asm` i `chkstk.asm` w `C:\Program Files (x86)\Microsoft Visual Studio 10.0\VC\crt\src\intel`

```
    push    1
    push    OFFSET $SG2672
    push    600         ; 00000258H
    push    esi
    call    __snprintf

    push    esi
    call    _puts
    add     esp, 28

...
```

Jedyny parametr `alloca()` jest przekazywany przez EAX (zamiast przez stos) [63]

GCC + składnia Intela

GCC 4.4.1 generuje podobne wyjście, ale bez wywoływania zewnętrznych funkcji:

Listing 1.41: GCC 4.7.3

```
.LC0:
        .string "hi! %d, %d, %d\n"
f:
        push    ebp
        mov     ebp, esp
        push    ebx
        sub     esp, 660
        lea     ebx, [esp+39]
        and     ebx, -16                   ; wyrównaj wskaźnik do granicy 16
    bajtów
        mov     DWORD PTR [esp], ebx       ; s
        mov     DWORD PTR [esp+20], 3
        mov     DWORD PTR [esp+16], 2
        mov     DWORD PTR [esp+12], 1
        mov     DWORD PTR [esp+8], OFFSET FLAT:.LC0 ; "hi! %d, %d, %d\n"
        mov     DWORD PTR [esp+4], 600     ; maxlen
        call    _snprintf
        mov     DWORD PTR [esp], ebx       ; s
        call    puts
        mov     ebx, DWORD PTR [ebp-4]
        leave
        ret
```

GCC + składnia AT&T

Spójrzmy na ten sam kod, ale w składni AT&T:

[63] Dlatego, że alloca() to nie tyle funkcja, co raczej *compiler intrinsic* (**??** on page ??). Jedną z przyczyn, dla której potrzebujemy osobnej funkcji a nie kilku instrukcji w samym kodzie, jest to, że implementacja alloca() z MSVC[64] zawiera kod, który czyta z właśnie zaalokowanej pamięci. W konsekwencji OS mapuje pamięć fizyczną na ten region pamięci wirtualnej. Po wywołaniu funkcji `alloca()` ESP pokazuje na blok o długości 600 bajtów, który można użyć na tablicę `buf`.

Listing 1.42: GCC 4.7.3

```
.LC0:
        .string "hi! %d, %d, %d\n"
f:
        pushl   %ebp
        movl    %esp, %ebp
        pushl   %ebx
        subl    $660, %esp
        leal    39(%esp), %ebx
        andl    $-16, %ebx
        movl    %ebx, (%esp)
        movl    $3, 20(%esp)
        movl    $2, 16(%esp)
        movl    $1, 12(%esp)
        movl    $.LC0, 8(%esp)
        movl    $600, 4(%esp)
        call    _snprintf
        movl    %ebx, (%esp)
        call    puts
        movl    -4(%ebp), %ebx
        leave
        ret
```

Kod jest taki sam jak na poprzednim listingu.

Nawiasem mówiąc, `movl $3, 20(%esp)` odpowiada `mov DWORD PTR [esp+20], 3` w składni Intela. W składni AT&T, sposób adresowania pamięci *rejestr+przesunięcie* zapisywany jest jako `przesunięcie(%rejestr)`.

(Windows) SEH

Na stosie są przechowywane wpisy SEH[65] dla funkcji (jeśli są one obecne). Więcej o tym tutaj: (5.2.1 on page 147).

Ochrona przed przepełnieniem bufora

Więcej o tym tutaj (1.19.2 on page 142).

Automatyczne zwalnianie miejsca na stosie

Zmienne lokalne i wpisy SEH są trzymane na stosie prawdopodobnie dlatego, że kiedy funkcja kończy działanie są one zwalniane automatycznie. Odbywa się to za pomocą tylko jednej instrukcji, zmieniającej wartość wskaźnika stosu — często jest to instrukcja ADD. Argumenty funkcji, można tak powiedzieć, również są automatycznie zwalniane na końcu funkcji. Z kolei wszystko, co jest przechowywane na stercie (*heap*), trzeba zwalniać jawnie.

[65] Structured Exception Handling

1.9.3 Struktura typowego stosu

Struktura typowego stosu w środowisku 32-bitowym, przed wykonaniem pierwszej instrukcji w funkcji, wygląda następująco:

…	…
ESP-0xC	zmienna lokalna#2, oznaczona w programie IDA jako var_8
ESP-8	zmienna lokalna#1, oznaczona w programie IDA jako var_4
ESP-4	odłożona wartość EBP
ESP	adres powrotu
ESP+4	argument#1, oznaczony w programie IDA jako arg_0
ESP+8	argument#2, oznaczony w programie IDA jako arg_4
ESP+0xC	argument#3, oznaczony w programie IDA jako arg_8
…	…

1.9.4 Śmieci na stosie

When one says that something seems random, what one usually means in practice is that one cannot see any regularities in it.

Stephen Wolfram, A New Kind of Science.

Często w tej książce mówimy o „szumie" lub „śmieciach" na stosie czy w pamięci. Skąd one się biorą? Są to pozostałości po poprzednich wywołaniach funkcji.

Krótki przykład:

```
#include <stdio.h>

void f1()
{
        int a=1, b=2, c=3;
};

void f2()
{
        int a, b, c;
        printf ("%d, %d, %d\n", a, b, c);
};

int main()
{
        f1();
        f2();
};
```

Kompilujemy…

Listing 1.43: Nieoptymalizujący MSVC 2010

```
$SG2752 DB      '%d, %d, %d', 0aH, 00H
```

```
_c$ = -12       ; size = 4
_b$ = -8        ; size = 4
_a$ = -4        ; size = 4
_f1     PROC
        push    ebp
        mov     ebp, esp
        sub     esp, 12
        mov     DWORD PTR _a$[ebp], 1
        mov     DWORD PTR _b$[ebp], 2
        mov     DWORD PTR _c$[ebp], 3
        mov     esp, ebp
        pop     ebp
        ret     0
_f1     ENDP

_c$ = -12       ; size = 4
_b$ = -8        ; size = 4
_a$ = -4        ; size = 4
_f2     PROC
        push    ebp
        mov     ebp, esp
        sub     esp, 12
        mov     eax, DWORD PTR _c$[ebp]
        push    eax
        mov     ecx, DWORD PTR _b$[ebp]
        push    ecx
        mov     edx, DWORD PTR _a$[ebp]
        push    edx
        push    OFFSET $SG2752 ; '%d, %d, %d'
        call    DWORD PTR __imp__printf
        add     esp, 16
        mov     esp, ebp
        pop     ebp
        ret     0
_f2     ENDP

_main   PROC
        push    ebp
        mov     ebp, esp
        call    _f1
        call    _f2
        xor     eax, eax
        pop     ebp
        ret     0
_main   ENDP
```

Kompilator się trochę oburzy…

```
c:\Polygon\c>cl st.c /Fast.asm /MD
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 16.00.40219.01 for ↙
    ↳ 80x86
Copyright (C) Microsoft Corporation.  All rights reserved.

st.c
```

```
c:\polygon\c\st.c(11) : warning C4700: uninitialized local variable 'c' ⤦
    ↳ used
c:\polygon\c\st.c(11) : warning C4700: uninitialized local variable 'b' ⤦
    ↳ used
c:\polygon\c\st.c(11) : warning C4700: uninitialized local variable 'a' ⤦
    ↳ used
Microsoft (R) Incremental Linker Version 10.00.40219.01
Copyright (C) Microsoft Corporation.  All rights reserved.

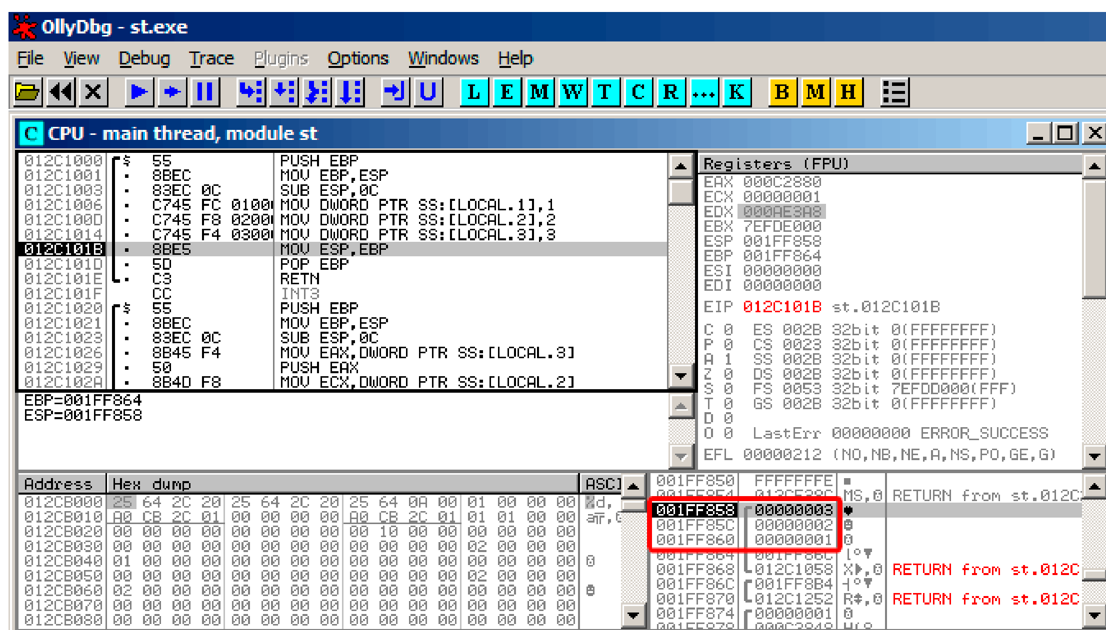
/out:st.exe
st.obj
```

Ale kiedy uruchomimy skompilowany program…

```
c:\Polygon\c>st
1, 2, 3
```

Dziwne, przecież nie ustawialiśmy żadnych zmiennych w `f2()`. Te wartości to „duchy”, które wciąż znajdują się na stosie.

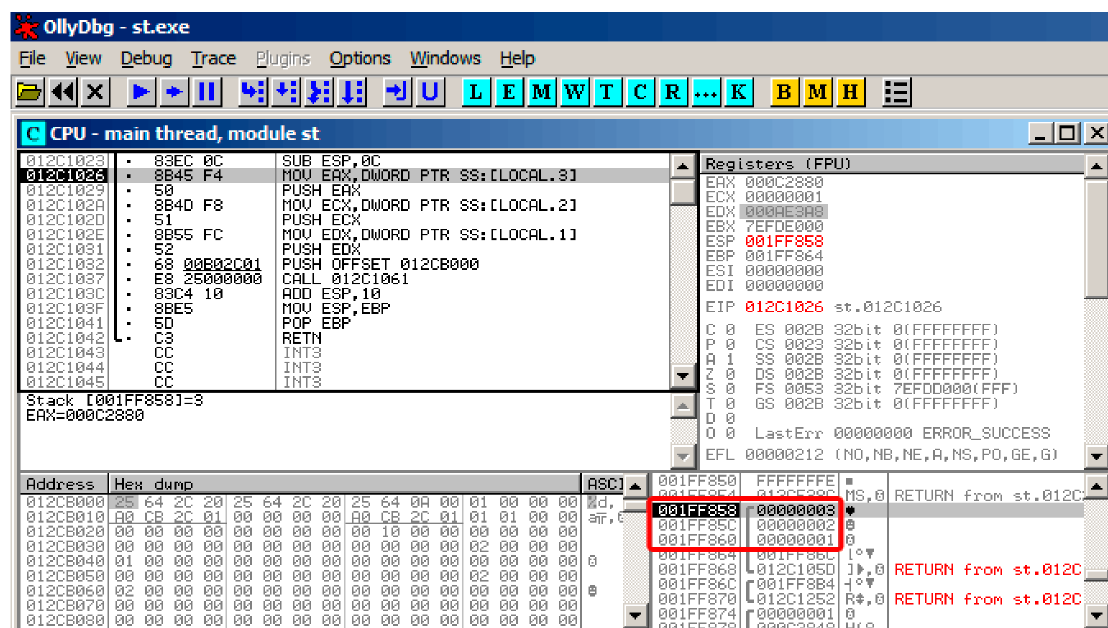
Spróbujmy uruchomić ten przykład w OllyDbg:



Rysunek 1.6: OllyDbg: f1()

Kiedy f1() ustawia zmienne a, b i c są one zapisywane pod adresem 0x1FF860, itd.

A kiedy jest wykonywana f2():



Rysunek 1.7: OllyDbg: f2()

... a, b i c w funkcji f2() znajdują się pod tymi samymi adresami! Nikt jeszcze nie nadpisał tych wartości, więc na razie pozostają one nietknięte. Taka dziwna sytuacja ma miejsce, kiedy kilka funkcji jest wykonywanych jedna po drugiej, a SP jest taki sam (funkcje mają taką samą liczbę argumentów). Wtedy zmienne lokalne będą przechowywane w tych samych adresach na stosie. Podsumowując, wszystkie wartości na stosie (i ogólnie w pamięci) to wartości pozostałe po poprzednich funkcjach. Nie są one losowe, w ścisłym tego słowa znaczeniu, lecz nieprzewidywalne. Czy można coś z tym zrobić? Można by czyścić fragmenty stosu przed wykonywaniem funkcji, ale to za dużo zbędnej (i nieporzebnej) pracy.

MSVC 2013

Przykład był skompilowany w MSVC 2010. Jeden czytelnik tej książki spróbował skompilować to w MSVC 2013, uruchomił i zobaczył 3 liczby w odwrotnej kolejności:

```
c:\Polygon\c>st
3, 2, 1
```

Dlaczego? Również spróbowałem skompilować ten przykład w MSVC 2013 i otrzymałem:

Listing 1.44: MSVC 2013

```
_a$ = -12        ; size = 4
```

```
_b$ = -8        ; size = 4
_c$ = -4        ; size = 4
_f2     PROC

...

_f2     ENDP

_c$ = -12       ; size = 4
_b$ = -8        ; size = 4
_a$ = -4        ; size = 4
_f1     PROC

...

_f1     ENDP
```

W odróżnieniu od MSVC 2010, MSVC 2013 rozmieścił zmienne a/b/c w funkcji `f2()` w odwrotnej kolejności. Jest to całkowicie poprawne, ponieważ w C/C++ nie ma zdefiniowanego standardu, który by wyznaczał kolejność zmiennych lokalnych na stosie. Przyczyną różnicy są zapewne zmiany w kodzie kompilatora, a więc nowsze MSVC zachowuje się nieco inaczej.

1.9.5 Ćwiczenia

- `http://challenges.re/51`
- `http://challenges.re/52`

1.10 Funkcja niemal pusta

Poniższy fragment kodu znalazłem w projekcie Boolector[66]:

```
// forward declaration. the function is residing in some other module:
int boolector_main (int argc, char **argv);

// executable
int main (int argc, char **argv)
{
        return boolector_main (argc, argv);
}
```

Dlaczego ktoś miałby tak robić? Prawdopodobnie `boolector_main()` może być kompilowane do biblioteki dynamicznej (jak np. DLL) i wywoływane w testach. Kod testowy również może przygotować argumenty `argc/argv`, tak jak to robi CRT.

Ciekawy jest wynik kompilacji:

[66] `https://boolector.github.io/`

Listing 1.45: Nieoptymalizujący GCC 8.2 x64 (wyjście w asemblerze)

```
main:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     DWORD PTR -4[rbp], edi
        mov     QWORD PTR -16[rbp], rsi
        mov     rdx, QWORD PTR -16[rbp]
        mov     eax, DWORD PTR -4[rbp]
        mov     rsi, rdx
        mov     edi, eax
        call    boolector_main
        leave
        ret
```

Mamy tutaj: prolog, niepotrzebne (niezoptymalizowane) przetasowanie dwóch argumentów, CALL, epilog i RET.

Zobaczmy na efekt kompilacji GCC z włączoną optymalizacją:

Listing 1.46: Optymalizujący GCC 8.2 x64 (wyjście w asemblerze)

```
main:
        jmp     boolector_main
```

Bardzo prosty kod — rejestr i stos zostały nienaruszone, gdyż `boolector_main()` ma taki sam zestaw argumentów. Jedyne co należało zrobić, to przekazać sterowanie pod inny adres.

Przypomina to thunk funkcje.

Później zobaczymy nieco bardziej zaawansowane przykłady: 1.11.2 on page 73, **??** on page ??.

1.11 printf() z wieloma argumentami

Spróbujmy rozszerzyć przykład *Hello, world!* (1.5 on page 12):

```
#include <stdio.h>

int main()
{
        printf("a=%d; b=%d; c=%d", 1, 2, 3);
        return 0;
};
```

1.11.1 x86

x86: 4 argumenty

MSVC

Gdy skompilujemy kod za pomocą MSVC 2010 Express, otrzymamy:

```
$SG3830 DB      'a=%d; b=%d; c=%d', 00H

...

        push    3
        push    2
        push    1
        push    OFFSET $SG3830
        call    _printf
        add     esp, 16                                 ; 00000010H
```

Kod jest niemal identyczny z tym, który widzieliśmy w *Hello, world!* (1.5 on page 12), lecz teraz argumenty funkcji `printf()` zostały odłożone na stos w odwrotnej kolejności. Pierwszy argument jest zapisywany jako ostatni.

Przy okazji, zmienna typu *int* w środowisku 32-bitowym ma długość 32 bitów, czyli 4 bajtów.

Mamy więc 4 argumenty. $4*4=16$ —zajmują dokładnie 16 bajtów na stosie: 32-bitowy wskaźnik na łańcuch znaków i trzy liczby typu *int*.

Gdy wskaźnik stosu (rejestr ESP) jest przywracany za pomocą `ADD ESP, X` za wywołaniem funkcji, to często można określić liczbę argumentów, dzieląc X przez 4.

Oczywiście dotyczy to tylko konwencji wywołań *cdecl* i środowiska 32-bitowego!

O konwencjach wywołań przeczytasz tutaj (**??** on page ??).

Kompilator, gdy kilka funkcji jest wywoływanych jedna za drugą, może połączyć kilka instrukcji „ADD ESP, X” w jedną i umieścić ją po ostatnim wywołaniu:

```
push a1
push a2
call ...
...
push a1
call ...
...
push a1
push a2
push a3
call ...
add esp, 24
```

Przykład prawdziwego kodu:

Listing 1.47: x86

```
.text:100113E7    push    3
.text:100113E9    call    sub_100018B0 ; wykorzystuje jeden argument (3)
.text:100113EE    call    sub_100019D0 ; funkcja bez argumentów
.text:100113F3    call    sub_10006A90 ; funkcja bez argumentów
.text:100113F8    push    1
.text:100113FA    call    sub_100018B0 ; wykorzystuje jeden argument (1)
```

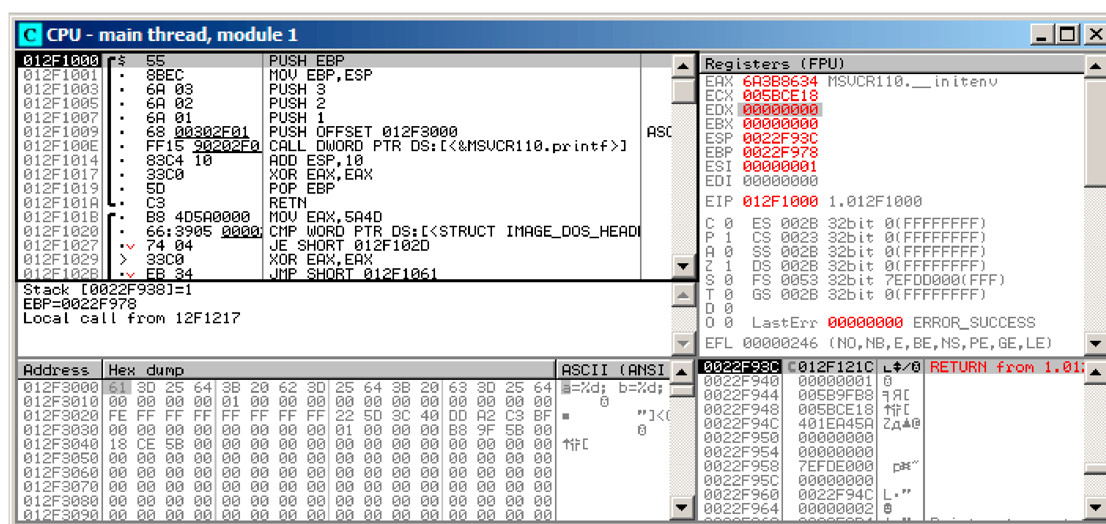
```
.text:100113FF    add     esp, 8       ; jednocześnie sprząta dwa argumenty
    ze stosu
```

MSVC and OllyDbg

Skorzystajmy z OllyDbg. Jest to jeden z najpopularniejszych debuggerów pod win32, działających w trybie użytkownika. Przykład skompilujemy w MSVC 2012 z opcją `/MD`, która oznacza dynamiczne linkowanie do `MSVCR*.DLL`, by łatwo można było rozpoznać zaimportowane funkcje w debuggerze.

Możemy załadować plik wykonywalny do OllyDbg. Pierwszy breakpoint jest w `ntdll.dll`, naciskamy F9 (run). Drugi breakpoint jest w kodzie CRT. Musimy odnaleźć funkcję `main()`.

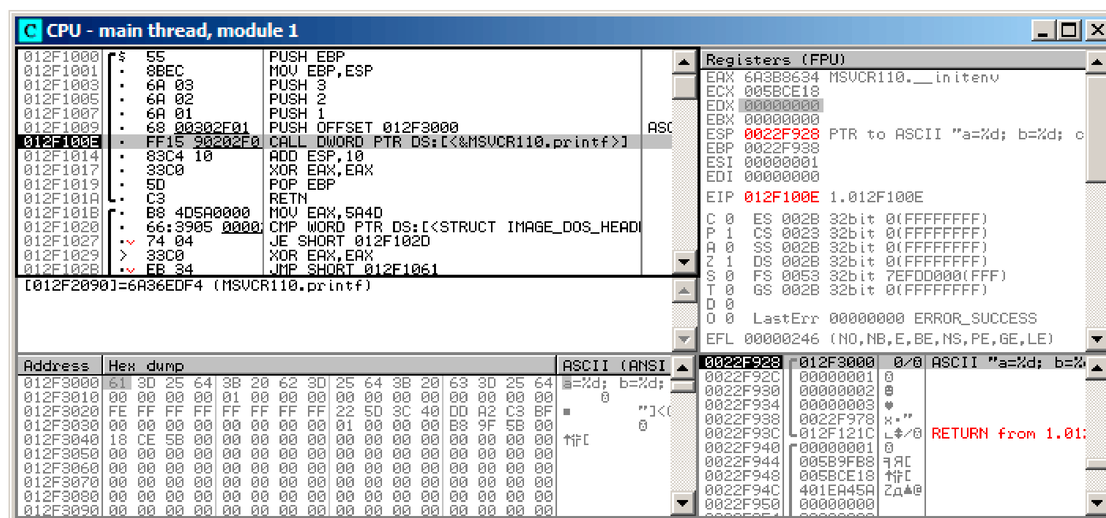
Można to zrobić scrollując na samą górę kodu (MSVC umieszcza funkcję `main()` na samym początku sekcji kodu):



Rysunek 1.8: OllyDbg: sam początek funkcji `main()`

Klikamy na instrukcję PUSH EBP, wciskamy F2 (ustaw breakpoint) i wciskamy F9 (run). Dzięki temu przeskoczymy kod z CRT, którym nie jesteśmy na razie zainteresowani.

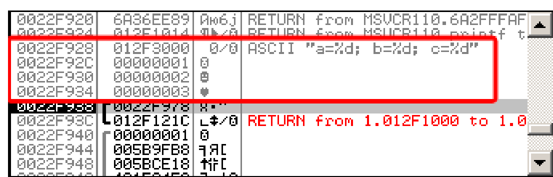
Naciśnij F8 (step over) 6 razy, by przeskoczyć 6 kolejnych instrukcji:



Rysunek 1.9: OllyDbg: przed wykonaniem `printf()`

PC pokazuje teraz na instrukcję CALL `printf`. OllyDbg, jak inne debuggery, podświetla rejestry, które zostały zmienione. Za każdym razem, gdy naciskasz F8, EIP zmienia się i jego wartość jest wyświetlana na czerwono. ESP również się zmienia, gdyż argumenty są przekazywane przez stos.

Gdzie na stosie są nasze wartości? Spójrz na obszar w prawym, dolnym rogu okna:



Rysunek 1.10: OllyDbg: stos po odłożeniu argumentów (czerwona ramka została dodana przez autora w programie graficznym)

Widzimy 3 kolumny: adres na stosie, wartość i dodatkowo komentarz OllyDbg. OllyDbg rozumie wskaźniki na łańcuchy znaków, więc wypisuje wartość tego łańcucha jako komentarz .

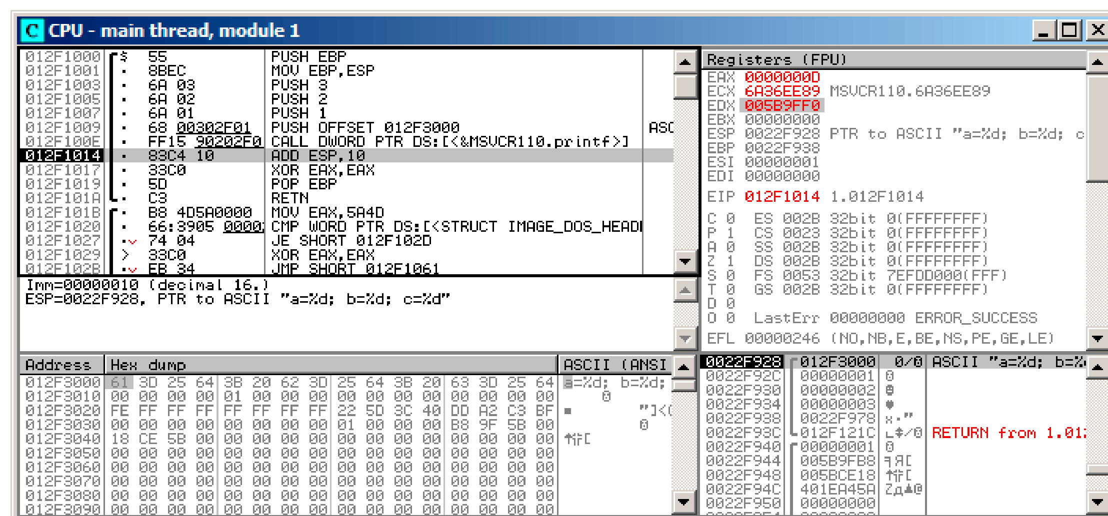
Możesz kliknąć prawym przyciskiem myszy na łańcuch znaków z formatem i wybrać „Follow in dump”. Łańcuch znaków pojawi się w lewym, dolnym oknie debuggera, wyświetlającym zawartość pamięci. Możesz ją tam edytować. Mógłbyś zmienić format łańcucha znaków, tak by na wyjście został wypisany inny tekst. W tym przykładzie nie jest to użyteczna funkcjonalność, ale możesz jej użyć w ramach ćwiczeń, by lepiej poznać opisane wcześniej mechanizmy.

Wciśnij F8 (step over).

Zobaczymy następujący rezultat w konsoli:

```
a=1; b=2; c=3
```

Sprawdźmy jak zmieniły się rejestry i stos:



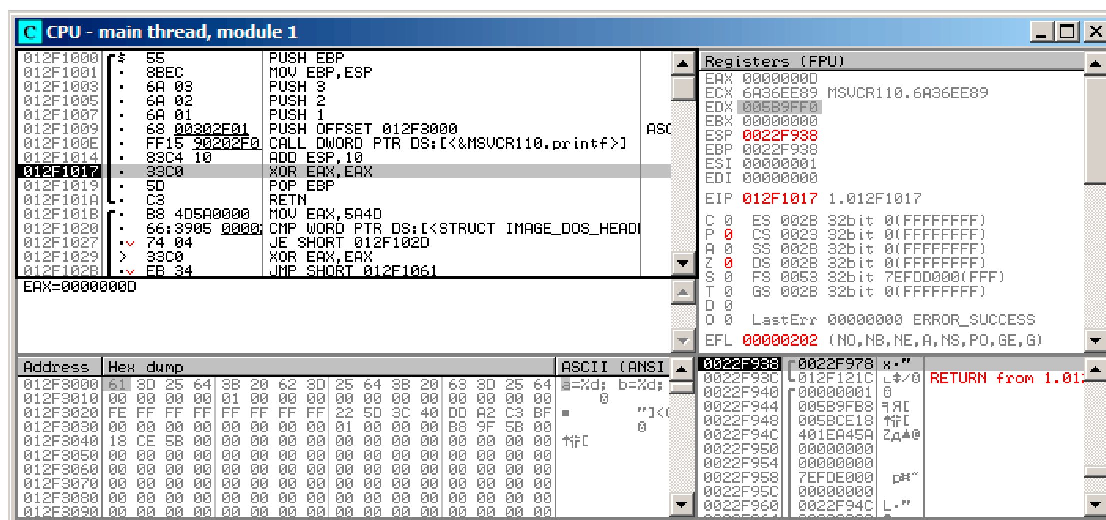
Rysunek 1.11: OllyDbg po wykonaniu funkcji `printf()`

Rejestr EAX zawiera teraz `0xD` (13). Jest to spodziewana wartość, ponieważ `printf()` zwraca liczbę wypisanych znaków. Zmieniła się wartość EIP: zawiera teraz adres kolejnej instrukcji, występującej bezpośrednio za `CALL printf`. ECX i EDX również się zmieniły. Najwyraźniej implementacja `printf()` wykorzystała je do własnych celów.

Ważnym faktem jest to, że ani wartość ESP, ani stan stosu się nie zmieniły!

Łatwo zauważyć, że łańcuch znaków z formatem oraz 3 powiązane z nim argumentu wciąż tam są. Jest to konwencja wywoływania *cdecl*: funkcja wywoływana nie przywraca ESP do pierwotnego stanu. Ta odpowiedzialność spoczywa na wywołującym.

Ponownie wciśnij F8, by wykonać instrukcję ADD ESP, 10:



Rysunek 1.12: OllyDbg: po wykonaniu instrukcji ADD ESP, 10

Zmieniła się wartość ESP, ale wartości na stosie zostały niezmienione! Można się tego spodziewać; nikt nie musi ich nadpisywać. Wszystko powyżej wskaźnika stosu (SP) to *szum* czy *śmieci* i nie ma znaczenia. Czyszczenie nieużywanych wpisów na stosie byłoby stratą czasu i nikt tego nie potrzebuje.

GCC

Skompilujmy ten sam program na Linuksie, za pomocą GCC 4.4.1 i sprawdźmy wynik w programie IDA:

```
main            proc near

var_10          = dword ptr -10h
var_C           = dword ptr -0Ch
var_8           = dword ptr -8
var_4           = dword ptr -4

                push    ebp
                mov     ebp, esp
                and     esp, 0FFFFFFF0h
                sub     esp, 10h
                mov     eax, offset aADBDCD ; "a=%d; b=%d; c=%d"
                mov     [esp+10h+var_4], 3
                mov     [esp+10h+var_8], 2
                mov     [esp+10h+var_C], 1
                mov     [esp+10h+var_10], eax
                call    _printf
```

```
                mov     eax, 0
                leave
                retn
main            endp
```

Jedyną zauważalną różnicą jest inny sposób odkładania argumentów na stos. W tym przykładzie GCC explicite podaje adres w obrębie stosu i nie używa instrukcji PUSH/POP.

GCC i GDB

Wczytajmy plik wykonywalny do debuggera GDB[67].

Opcja -g sprawia, że kompilator zapisuje do pliku wykonywalnego informacje użyteczne do debuggowania.

```
$ gcc 1.c -g -o 1
```

```
$ gdb 1
GNU gdb (GDB) 7.6.1-ubuntu
...
Reading symbols from /home/dennis/polygon/1...done.
```

Listing 1.48: Ustawmy breakpoint w funkcji `printf()`

```
(gdb) b printf
Breakpoint 1 at 0x80482f0
```

Kontynuujmy wykonywanie kodu. Gdybyśmy dysponowali kodem źródłowym funkcji `printf()`, GDB mógłby go wyświetlić obok instrukcji asemblera.

```
(gdb) run
Starting program: /home/dennis/polygon/1

Breakpoint 1, __printf (format=0x80484f0 "a=%d; b=%d; c=%d") at printf.c:29
29      printf.c: No such file or directory.
```

Wypiszmy 10 wartości ze stosu. Skrajna lewa kolumna oznacza adres, pod którym wartości ze stosu znajdują się w pamięci.

```
(gdb) x/10w $esp
0xbffff11c:     0x0804844a      0x080484f0      0x00000001      0x00000002
0xbffff12c:     0x00000003      0x08048460      0x00000000      0x00000000
0xbffff13c:     0xb7e29905      0x00000001
```

Pierwszy element to adres powrotu (RA) (`0x0804844a`). Możemy to sprawdzić, deasemblując pamięć pod tym adresem (instruujemy GDB, by wypisał 5 elementów ze stosu, interpretując je jako instrukcje asemblera):

[67] GNU Debugger

```
(gdb) x/5i 0x0804844a
   0x804844a <main+45>: mov    $0x0,%eax
   0x804844f <main+50>: leave
   0x8048450 <main+51>: ret
   0x8048451:   xchg   %ax,%ax
   0x8048453:   xchg   %ax,%ax
```

Pojawiająca się dwa razy XCHG działa jak pusta instrukcja NOP, gdyż w tym przypadku ustawia wartość rejestr AX na jego bieżącą wartość.

Drugim elementem jest (0x080484f0) — to adres łańcucha znaków z formatem:

```
(gdb) x/s 0x080484f0
0x80484f0:      "a=%d; b=%d; c=%d"
```

Kolejne 3 elementy (1, 2, 3) to argumenty funkcji `printf()`. Pozostałe elementy to prawdopodobnie „śmieci" znajdujące się na stosie, lecz mogłyby to być wartości używane przez inne funkcje, ich zmienne lokalnce, etc. Możemy je na razie zignorować.

Kontynuujmy za pomocą „finish". Polecenie powoduje, że GDB wykona wszystkie instrukcje aż do końca funkcji. W tym przypadku do końca funkcji `printf()`.

```
(gdb) finish
Run till exit from #0  __printf (format=0x80484f0 "a=%d; b=%d; c=%d") at ⤦
    ↳ printf.c:29
main () at 1.c:6
6               return 0;
Value returned is $2 = 13
```

GDB pokazuje jaką wartość `printf()` zwróciła przez rejestr EAX (13). Jest to liczba znaków, które zostały wypisane i jest to dokładnie tyle, ile widzieliśmy w przykładzie z OllyDbg.

Widzimy również „return 0;" wraz z informacją, że to wyrażenie jest w pliku `1.c`, w linii 6. Plik `1.c` znajduje się w bieżącym katalogu i tam GDB znalazł ten łańcuch znaków. Skąd debugger wiedział, która linia kodu w C jest właśnie wykonywana? Kompilatory, gdy generują informacje dla debuggera, zapisują również tablicę z zależnościami między liniami kodu źródłowego a adresami instrukcji, GDB jest w końcu debuggerem pracującym z kodem źródłowym.

Przyjrzyjmy się rejestrom. W EAX znajduje się wartość 13:

```
(gdb) info registers
eax            0xd      13
ecx            0x0      0
edx            0x0      0
ebx            0xb7fc0000       -1208221696
esp            0xbffff120       0xbffff120
ebp            0xbffff138       0xbffff138
esi            0x0      0
edi            0x0      0
eip            0x804844a        0x804844a <main+45>
...
```

Deasemblujemy kolejne instrukcje. Strzałka pokazuje na instrukcję, która zostanie wykonana jako kolejna.

```
(gdb) disas
Dump of assembler code for function main:
   0x0804841d <+0>:     push   %ebp
   0x0804841e <+1>:     mov    %esp,%ebp
   0x08048420 <+3>:     and    $0xfffffff0,%esp
   0x08048423 <+6>:     sub    $0x10,%esp
   0x08048426 <+9>:     movl   $0x3,0xc(%esp)
   0x0804842e <+17>:    movl   $0x2,0x8(%esp)
   0x08048436 <+25>:    movl   $0x1,0x4(%esp)
   0x0804843e <+33>:    movl   $0x80484f0,(%esp)
   0x08048445 <+40>:    call   0x80482f0 <printf@plt>
=> 0x0804844a <+45>:    mov    $0x0,%eax
   0x0804844f <+50>:    leave
   0x08048450 <+51>:    ret
End of assembler dump.
```

GDB używa domyślnie składni AT&T. Możemy jednak przełączyć się na składnię Intela.

```
(gdb) set disassembly-flavor intel
(gdb) disas
Dump of assembler code for function main:
   0x0804841d <+0>:     push   ebp
   0x0804841e <+1>:     mov    ebp,esp
   0x08048420 <+3>:     and    esp,0xfffffff0
   0x08048423 <+6>:     sub    esp,0x10
   0x08048426 <+9>:     mov    DWORD PTR [esp+0xc],0x3
   0x0804842e <+17>:    mov    DWORD PTR [esp+0x8],0x2
   0x08048436 <+25>:    mov    DWORD PTR [esp+0x4],0x1
   0x0804843e <+33>:    mov    DWORD PTR [esp],0x80484f0
   0x08048445 <+40>:    call   0x80482f0 <printf@plt>
=> 0x0804844a <+45>:    mov    eax,0x0
   0x0804844f <+50>:    leave
   0x08048450 <+51>:    ret
End of assembler dump.
```

Wykonajmy kolejną linię kodu w C/C++. GDB pokazuje klamrowy nawias zamykający, który oznacza koniec bloku kodu.

```
(gdb) step
7       };
```

Sprawdźmy rejestry po wykonaniu instrukcji `MOV EAX, 0`. Tak jak się spodziewamy, EAX zawiera teraz wartość 0.

```
(gdb) info registers
eax            0x0      0
ecx            0x0      0
edx            0x0      0
ebx            0xb7fc0000       -1208221696
```

```
esp            0xbffff120       0xbffff120
ebp            0xbffff138       0xbffff138
esi            0x0      0
edi            0x0      0
eip            0x804844f        0x804844f <main+50>
...
```

x64: 9 argumentów

Zmieńmy nieco nasz przykład, by zobaczyć jak na x64 argumenty przekazywane są przez stos. Zwiększymy liczbę argumentów do 9 (łańcuch znaków z formatem + 8 zmiennych typu *int*):

```
#include <stdio.h>

int main()
{
        printf("a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n", 1, 2, 3,↙
    ↘  4, 5, 6, 7, 8);
        return 0;
};
```

MSVC

Jak wspomnieliśmy poprzednio — pierwsze 4 argumenty na Win64 przekazywane są przez rejestry RCX, RDX, R8 i R9 a pozostałe przez stos[68].

Widać to na wygenerowanym listingu. Stos został przygotowany przy pomocy instrukcji MOV, zamiast PUSH- wartości zostały odłożone ze wskazaniem adresu.

Listing 1.49: MSVC 2012 x64

```
$SG2923 DB      'a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d', 0aH, 00H

main    PROC
        sub     rsp, 88

        mov     DWORD PTR [rsp+64], 8
        mov     DWORD PTR [rsp+56], 7
        mov     DWORD PTR [rsp+48], 6
        mov     DWORD PTR [rsp+40], 5
        mov     DWORD PTR [rsp+32], 4
        mov     r9d, 3
        mov     r8d, 2
        mov     edx, 1
        lea     rcx, OFFSET FLAT:$SG2923
        call    printf
```

[68]przyp. tłum. - rzecz wygląda inaczej w przypadku przekazywania argumentów zmiennoprzecinkowych, gdy może zostać wykorzystany rejestr wektorowy XMM0-XMM3

```
        ; zwróć 0
        xor     eax, eax

        add     rsp, 88
        ret     0
main    ENDP
_TEXT   ENDS
END
```

Uważny czytelnik może się zastanawiać, dlaczego dla wartości typu *int* zostało zaalokowanych 8 bajtów, skoro wystarczą tylko 4? Dla przypomnienia: 8 bajtów jest alokowanych dla każdego typu danych, krótszego niż 64 bity. Powodem jest wygoda, łatwo w ten sposób wyliczyć adres dowolnego argumentu. Poza tym, wszystkie są przechowywane w pamięci na wyrównanych adresach. W środowisku 32-bitowym jest podobnie - 4 bajty są zarezerwowane dla wszystkich typów nie dłuższych niż 4 bajty[69].

GCC

Kompilacja na x86-64 systemach *NIX daje podobny wynik jak MSVC, jednak teraz 6 pierwszych argumentów jest przekazywanych przez rejestry RDI, RSI, RDX, RCX, R8 i R9, a cała reszta przez stos.

GCC generuje kod, który przechowuje wskaźnik stosu w EDI zamiast w RDI — co już zdążyliśmy zauważyć: 1.5.2 on page 22.

Widzieliśmy też, że rejestr EAX został wyzerowany przed wywołaniem funkcji `printf()`: 1.5.2 on page 22.

Listing 1.50: Optymalizujący GCC 4.4.6 x64

```
.LC0:
        .string "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n"

main:
        sub     rsp, 40

        mov     r9d, 5
        mov     r8d, 4
        mov     ecx, 3
        mov     edx, 2
        mov     esi, 1
        mov     edi, OFFSET FLAT:.LC0
        xor     eax, eax ; liczba użytych rejestrów wektorowych
        mov     DWORD PTR [rsp+16], 8
        mov     DWORD PTR [rsp+8], 7
        mov     DWORD PTR [rsp], 6
        call    printf

        ; zwróć 0

        xor     eax, eax
```

[69]przyp. tłum. - np. zmienna typu long long zajmie 8 bajtów.

```
        add     rsp, 40
        ret
```

GCC + GDB

Wczytajmy plik wykonywalny do debuggera GDB

```
$ gcc -g 2.c -o 2
```

```
$ gdb 2
GNU gdb (GDB) 7.6.1-ubuntu
...
Reading symbols from /home/dennis/polygon/2...done.
```

Listing 1.51: Ustawiamy breakpoint na funkcji `printf()` i zaczynamy wykonanie

```
(gdb) b printf
Breakpoint 1 at 0x400410
(gdb) run
Starting program: /home/dennis/polygon/2

Breakpoint 1, __printf (format=0x400628 "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d⤦
    ↳ ; g=%d; h=%d\n") at printf.c:29
29      printf.c: No such file or directory.
```

Rejestry RSI/RDX/RCX/R8/R9 zostały ustawione na spodziewane wartości. W RIP przechowywany jest adres pierwszej instrukcji z funkcji `printf()`.

```
(gdb) info registers
rax            0x0      0
rbx            0x0      0
rcx            0x3      3
rdx            0x2      2
rsi            0x1      1
rdi            0x400628 4195880
rbp            0x7fffffffdf60   0x7fffffffdf60
rsp            0x7fffffffdf38   0x7fffffffdf38
r8             0x4      4
r9             0x5      5
r10            0x7fffffffdce0   140737488346336
r11            0x7ffff7a65f60   140737348263776
r12            0x400440 4195392
r13            0x7fffffffe040   140737488347200
r14            0x0      0
r15            0x0      0
rip            0x7ffff7a65f60   0x7ffff7a65f60 <__printf>
...
```

Listing 1.52: Łańcuch znaków z formatem

```
(gdb) x/s $rdi
0x400628:       "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n"
```

Wyświetlimy zawartość stosu za pomocą komendy x/g —*g* oznacza *giant word*, czyli wartość 64-bitową.

```
(gdb) x/10g $rsp
0x7fffffffdf38: 0x0000000000400576      0x0000000000000006
0x7fffffffdf48: 0x0000000000000007      0x00007fff00000008
0x7fffffffdf58: 0x0000000000000000      0x0000000000000000
0x7fffffffdf68: 0x00007ffff7a33de5      0x0000000000000000
0x7fffffffdf78: 0x00007fffffffe048      0x0000000100000000
```

Jak w poprzednim przykładzie, pierwszy elementem na stosie jest adres powrotu RA. 3 wartości przekazywane są przez stos: 6, 7, 8. Widać, że 8 przekazywane jest z wypełnionymi 32 starszymim bitami: `0x00007fff00000008`. Nie stanowi to problemu, ponieważ argument jest typu *int*, który jest 32-bitowy. Starsze części rejestrów, albo elementów na stosie, mogą zawierać „losowe śmieci”.

Jeśli sprawdzimy gdzie zostanie zwrócone sterowanie po zakończeniu funkcji `printf()`, GDB pokaże funkcję main:

```
(gdb) set disassembly-flavor intel
(gdb) disas 0x0000000000400576
Dump of assembler code for function main:
   0x000000000040052d <+0>:     push   rbp
   0x000000000040052e <+1>:     mov    rbp,rsp
   0x0000000000400531 <+4>:     sub    rsp,0x20
   0x0000000000400535 <+8>:     mov    DWORD PTR [rsp+0x10],0x8
   0x000000000040053d <+16>:    mov    DWORD PTR [rsp+0x8],0x7
   0x0000000000400545 <+24>:    mov    DWORD PTR [rsp],0x6
   0x000000000040054c <+31>:    mov    r9d,0x5
   0x0000000000400552 <+37>:    mov    r8d,0x4
   0x0000000000400558 <+43>:    mov    ecx,0x3
   0x000000000040055d <+48>:    mov    edx,0x2
   0x0000000000400562 <+53>:    mov    esi,0x1
   0x0000000000400567 <+58>:    mov    edi,0x400628
   0x000000000040056c <+63>:    mov    eax,0x0
   0x0000000000400571 <+68>:    call   0x400410 <printf@plt>
   0x0000000000400576 <+73>:    mov    eax,0x0
   0x000000000040057b <+78>:    leave
   0x000000000040057c <+79>:    ret
End of assembler dump.
```

Przeskoczmy na koniec funkcji `printf()` i wykonajmy jedną linię kodu z funkcji `main()`. W ramach tej linii rejestr EAX został wyzerowany. Debugger pokazuje teraz na koniec bloku kodu. Możemy się upewnić, że EAX rzeczywiście ma wartość 0, a RIP pokazuje teraz na instrukcję LEAVE, przedostatnią w funkcji `main()`.

```
(gdb) finish
Run till exit from #0  __printf (format=0x400628 "a=%d; b=%d; c=%d; d=%d; e↙
    ↳ =%d; f=%d; g=%d; h=%d\n") at printf.c:29
a=1; b=2; c=3; d=4; e=5; f=6; g=7; h=8
main () at 2.c:6
6               return 0;
Value returned is $1 = 39
```

```
(gdb) next
7       };
(gdb) info registers
rax            0x0      0
rbx            0x0      0
rcx            0x26     38
rdx            0x7ffff7dd59f0   140737351866864
rsi            0x7fffffd9       2147483609
rdi            0x0      0
rbp            0x7fffffffdf60   0x7fffffffdf60
rsp            0x7fffffffdf40   0x7fffffffdf40
r8             0x7ffff7dd26a0   140737351853728
r9             0x7ffff7a60134   140737348239668
r10            0x7fffffffd5b0   140737488344496
r11            0x7ffff7a95900   140737348458752
r12            0x400440 4195392
r13            0x7fffffffe040   140737488347200
r14            0x0      0
r15            0x0      0
rip            0x40057b 0x40057b <main+78>
...
```

1.11.2 ARM

ARM: 4 argumenty

Tradycyjny sposób w ARM na przekazywanie argumentów (tzw. konwencja wywoływania funkcji) wygląda następująco: pierwsze 4 argumenty są przekazywane przez rejestry R0-R3, a pozostałe przez stos. Przypomina to konwencję fastcall (**??** on page ??) czy win64 (**??** on page ??).

32-bitowy ARM

Nieoptymalizujący Keil 6/2013 (tryb ARM)

Listing 1.53: Nieoptymalizujący Keil 6/2013 (tryb ARM)

```
.text:00000000 main
.text:00000000 10 40 2D E9   STMFD   SP!, {R4,LR}
.text:00000004 03 30 A0 E3   MOV     R3, #3
.text:00000008 02 20 A0 E3   MOV     R2, #2
.text:0000000C 01 10 A0 E3   MOV     R1, #1
.text:00000010 08 00 8F E2   ADR     R0, aADBDCD     ; "a=%d; b=%d; c=%d"
.text:00000014 06 00 00 EB   BL      __2printf
.text:00000018 00 00 A0 E3   MOV     R0, #0          ; zwróć 0
.text:0000001C 10 80 BD E8   LDMFD   SP!, {R4,PC}
```

Pierwsze 4 argumenty zostały przekazane przez rejestry R0-R3 w następującej kolejności: wskaźnik na łańcuch znaków z formatem w R0, następnie 1 w R1, 2 w R2 i 3 w

R3. Instrukcja z adresu 0x18 zapisuje 0 do rejestru R0—jest to część instrukcji *return 0* z kodu C. Nic nadzwyczajnego.

Optymalizujący Keil 6/2013 generuje taki sam kod.

Optymalizujący Keil 6/2013 (tryb Thumb)

Listing 1.54: Optymalizujący Keil 6/2013 (tryb Thumb)

```
.text:00000000 main
.text:00000000 10 B5        PUSH    {R4,LR}
.text:00000002 03 23        MOVS    R3, #3
.text:00000004 02 22        MOVS    R2, #2
.text:00000006 01 21        MOVS    R1, #1
.text:00000008 02 A0        ADR     R0, aADBDCD     ; "a=%d; b=%d; c=%d"
.text:0000000A 00 F0 0D F8  BL      __2printf
.text:0000000E 00 20        MOVS    R0, #0
.text:00000010 10 BD        POP     {R4,PC}
```

Porównując do niezoptymalizowanego kodu w trybie ARM, nie widać wyraźnej różnicy.

Optymalizujący Keil 6/2013 (tryb ARM) + usunięty return

Zmieńmy nieco przykład, usuwając *return 0*:

```
#include <stdio.h>

void main()
{
        printf("a=%d; b=%d; c=%d", 1, 2, 3);
};
```

Efekt jest dość nieoczekiwany:

Listing 1.55: Optymalizujący Keil 6/2013 (tryb ARM)

```
.text:00000014 main
.text:00000014 03 30 A0 E3   MOV     R3, #3
.text:00000018 02 20 A0 E3   MOV     R2, #2
.text:0000001C 01 10 A0 E3   MOV     R1, #1
.text:00000020 1E 0E 8F E2   ADR     R0, aADBDCD     ; "a=%d; b=%d; c=%d\n"
.text:00000024 CB 18 00 EA   B       __2printf
```

W zoptymalizowanej wersji w trybie ARM widzimy, że ostatnią instrukcją jest B zamiast spodziewanej BL. Inną różnicą jest brak prologu i epilogu (instrukcje zachowujące wartości rejestrów R0 i LR), które wystąpiły w wersji niezoptymalizowanej.

Instrukcja B skacze pod inny adres, bez zmiany rejestru LR, podobnie jak JMP w x86. Dlaczego to działa? Nowy kod w działaniu jest równoważny poprzedniej wersji z dwóch powodów:

1) nie jest modyfikowany SP (wskaźnik stosu),

2) wywołanie `printf()` jest ostatnią instrukcją, nic się dalej nie dzieje.

Funkcja `printf()` po zakończeniu pracy zwraca sterowanie do adresu z LR. LR przechowuje adres miejsca, z którego nasza funkcja została wywołana, a więc tam też zostanie zwrócone sterowanie. Nie musimy zapisywać LR, ponieważ nie ma konieczności jego modyfikacji. W programie nie ma innych wywołań funkcji niż wywołanie `printf()` i to z tego powodu nie musimy modyfikować LR. Co więcej, po wywołaniu funkcji nie musimy już nic robić! To własnie dzięki tym wszystkim okolicznościom optymalizacja była możliwa.

Podobna optymalizacja pojawia się często w funkcjach, których ostatnią instrukcją jest wywołanie innej funkcji. Podobny przykład widać tutaj: **??** on page ??.

Nieco prostszy przykład opisywaliśmy już wcześniej: 1.10 on page 57.

ARM64

Nieoptymalizujący GCC (Linaro) 4.9

Listing 1.56: Nieoptymalizujący GCC (Linaro) 4.9

```
.LC1:
        .string "a=%d; b=%d; c=%d"
f2:
; zapisz FP i LR w ramce stosu:
        stp     x29, x30, [sp, -16]!
; ustaw wskaźnik ramki stosu (FP=SP):
        add     x29, sp, 0
        adrp    x0, .LC1
        add     x0, x0, :lo12:.LC1
        mov     w1, 1
        mov     w2, 2
        mov     w3, 3
        bl      printf
        mov     w0, 0
; przywróć FP i LR
        ldp     x29, x30, [sp], 16
        ret
```

Pierwsza instrukcja STP (*Store Pair*) zapisuje na stosie FP (X29) i LR (X30). Kolejna — `ADD X29, SP, 0` — tworzy ramkę stosu przez zapisanie wartości SP do X29.

Następnie widać już znaną parę instrukcji ADRP/ADD, która konstruuje wskaźnik na łańcuch znaków. *lo12* oznacza młodsze 12 bitów — linker umieści młodsze 12 bitów adresu LC1 w kodzie operacji (opcode) instrukcji ADD. Trzy ostatnie argumenty funkcji `printf()` — 1, 2 i 3 — to literały typu *int*, więc są ładowane do 32-bitowych części rejestru. [70]

[70] Zmiana 1 na 1L (literał `long long`) spowoduje, że będzie to wartość 64-bitowa i trafi ona do rejestru 64-bitowego. Więcej o definiowaniu liczb całkowitych i literałach w C/C++: 1, 2.

Optymalizujący GCC (Linaro) 4.9 generuje taki sam kod.

ARM: 9 argumentów

Użyjmy ponownie przykładu z 9 argumentami: 1.11.1 on page 68.

```
#include <stdio.h>

int main()
{
        printf("a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n", 1, 2, 3,↙
    ↳  4, 5, 6, 7, 8);
        return 0;
};
```

Optymalizujący Keil 6/2013: tryb ARM

```
.text:00000028             main
.text:00000028
.text:00000028             var_18 = -0x18
.text:00000028             var_14 = -0x14
.text:00000028             var_4  = -4
.text:00000028
.text:00000028 04 E0 2D E5  STR    LR, [SP,#var_4]!
.text:0000002C 14 D0 4D E2  SUB    SP, SP, #0x14
.text:00000030 08 30 A0 E3  MOV    R3, #8
.text:00000034 07 20 A0 E3  MOV    R2, #7
.text:00000038 06 10 A0 E3  MOV    R1, #6
.text:0000003C 05 00 A0 E3  MOV    R0, #5
.text:00000040 04 C0 8D E2  ADD    R12, SP, #0x18+var_14
.text:00000044 0F 00 8C E8  STMIA  R12, {R0-R3}
.text:00000048 04 00 A0 E3  MOV    R0, #4
.text:0000004C 00 00 8D E5  STR    R0, [SP,#0x18+var_18]
.text:00000050 03 30 A0 E3  MOV    R3, #3
.text:00000054 02 20 A0 E3  MOV    R2, #2
.text:00000058 01 10 A0 E3  MOV    R1, #1
.text:0000005C 6E 0F 8F E2  ADR    R0, aADBDCDDDEDFDGD ; "a=%d; b=%d; c=%d;
    d=%d; e=%d; f=%d; g=%"...
.text:00000060 BC 18 00 EB  BL     __2printf
.text:00000064 14 D0 8D E2  ADD    SP, SP, #0x14
.text:00000068 04 F0 9D E4  LDR    PC, [SP+4+var_4],#4
```

Kod można podzielić na kilka części.

- Prolog.

  Pierwsza instrukcja — `STR LR, [SP,#var_4]!` — zapisuje LR na stosie, ponieważ użyjemy tego rejestru przy wywołaniu funkcji `printf()`. Wykrzyknik na końcu oznacza *pre-index*.

Pre-index oznacza, że SP zostanie najpierw zmniejszony o 4, a następnie pod tym zmodyfikowanym adresem zostanie zapisana wartość LR. Podobnie działa instrukcja PUSH na x86. Więcej przeczytasz o tym tutaj: **??** on page ??.

Druga instrukcja — `SUB SP, SP, #0x14` — zmniejsza SP (wskaźnik stosu) by zaalokować na stosie `0x14` (20) bajtów. Będziemy przekazywać do funkcji `printf()` 5 32-bitowych wartości, każda z nich zajmuje 4 bajty — razem zajmą więc dokładnie 20 bajtów. Pozostałe 4 32-bitowe wartości zostaną przekazane przez rejestry.

- Przekazania argumentów 5, 6, 7 i 8 przez stos.

  Najpierw argumenty zapisywane są do rejestrów, odpowiednio: `R0`, `R1`, `R2` and `R3`. Następnie instrukcja `ADD R12, SP, #0x18+var_14` zapisuje do rejestru `R12` adres stosu, pod którym te wartości zostaną umieszczone. *var_14* to makro równe -0x14, stworzone przez program IDA by poprawić czytelność kodu pracującego ze stosem. Makra *var_?* wygenerowane w programie IDA odpowiadają zmiennym lokalnym umieszczonym na stosie. Zatem ostatecznie do rejestru R12 trafi adres SP+4.

  Kolejna instrukcja — `STMIA R12, R0-R3` — zapisuje zawartość rejestrów `R0-R3` w pamięci, pod adres z `R12`. `STMIA` to skrót od *Store Multiple Increment After*. *Increment After* oznacza, że `R12` będzie zwiększany o 4, po każdej zapisanej wartości rejestru.

- Przekazanie argumentu 4 przez stos.

  Najpierw wartość 4 jest zapisywana w `R0`, a następnie za pomocą instrukcji `STR R0, [SP,#0x18+var_18]` odkładana jest na stos. *var_18* to -0x18, a więc ostateczne przesunięcie (offset) wynosi 0, a więc wartość z `R0` (4) trafi pod adres z SP.

- Przekazanie argumentów 1, 2, 3 przez rejestry.

  3 pierwsze argumenty — 1, 2, 3 (a, b, c w łańcuchu formatującym) — są przekazywane przez rejestry R1, R2 i R3 tuż przed wywołaniem funkcji `printf()`.

- Wywołanie `printf()`.
- Epilog.

  Instrukcja `ADD SP, SP, #0x14` przywraca wskaźnik stosu SP do poprzedniej wartości, porzucając wszystkie tam zapisane dane. Oczywiście odłożone wartości wciąż tam są, ale zostaną nadpisane przy wywołaniach kolejnych funkcji.

  Instrukcja `LDR PC, [SP+4+var_4],#4` wczytuje do PC wcześniej odłożoną na stos wartość LR, powodując wyjście z funkcji. Tym razem na końcu instrukcji nie ma wykrzyknika — tak, najpierw PC jest ładowany z adresu przechowywanego w SP ($4 + var_4 = 4 + (-4) = 0$), a następnie SP jest zwiększany o 4.

  Dlaczego IDA w taki sposób wyświetla instrukcje? W ten sposób łatwiej pokazać układ danych na stosie, widać tutaj, że zmienna `var_4` została stworzona do zapisu wartości LR na stosie lokalnym. Instrukcje jest na swój sposób podobna do `POP PC` w x86[71].

[71]Niemożliwe jest ustawienie wartości w IP/EIP/RIP za pomocą POP w x86, ale poza tym to trafne

Optymalizujący Keil 6/2013: tryb Thumb

```
.text:0000001C             printf_main2
.text:0000001C
.text:0000001C             var_18 = -0x18
.text:0000001C             var_14 = -0x14
.text:0000001C             var_8  = -8
.text:0000001C
.text:0000001C 00 B5        PUSH    {LR}
.text:0000001E 08 23        MOVS    R3, #8
.text:00000020 85 B0        SUB     SP, SP, #0x14
.text:00000022 04 93        STR     R3, [SP,#0x18+var_8]
.text:00000024 07 22        MOVS    R2, #7
.text:00000026 06 21        MOVS    R1, #6
.text:00000028 05 20        MOVS    R0, #5
.text:0000002A 01 AB        ADD     R3, SP, #0x18+var_14
.text:0000002C 07 C3        STMIA   R3!, {R0-R2}
.text:0000002E 04 20        MOVS    R0, #4
.text:00000030 00 90        STR     R0, [SP,#0x18+var_18]
.text:00000032 03 23        MOVS    R3, #3
.text:00000034 02 22        MOVS    R2, #2
.text:00000036 01 21        MOVS    R1, #1
.text:00000038 A0 A0        ADR     R0, aADBDCDDDEDFDGD ; "a=%d; b=%d; c=%d;
    d=%d; e=%d; f=%d; g=%"...
.text:0000003A 06 F0 D9 F8  BL      __2printf
.text:0000003E
.text:0000003E             loc_3E   ; CODE XREF: example13_f+16
.text:0000003E 05 B0        ADD     SP, SP, #0x14
.text:00000040 00 BD        POP     {PC}
```

Wyjście jest podobne do poprzedniego przykładu. Tym razem jest to kod w trybie Thumb i wartości są umieszczane na stosie w innym porządku: najpierw odkładana jest wartość 8, jako druga odkładana jest grupa wartości 5, 6, 7 a jako trzecia odkładana jest wartość 4.

Optymalizujący Xcode 4.6.3 (LLVM): tryb ARM

```
__text:0000290C             _printf_main2
__text:0000290C
__text:0000290C             var_1C = -0x1C
__text:0000290C             var_C  = -0xC
__text:0000290C
__text:0000290C 80 40 2D E9   STMFD  SP!, {R7,LR}
__text:00002910 0D 70 A0 E1   MOV    R7, SP
__text:00002914 14 D0 4D E2   SUB    SP, SP, #0x14
__text:00002918 70 05 01 E3   MOV    R0, #0x1570
__text:0000291C 07 C0 A0 E3   MOV    R12, #7
__text:00002920 00 00 40 E3   MOVT   R0, #0
__text:00002924 04 20 A0 E3   MOV    R2, #4
```

porównanie.

```
__text:00002928 00 00 8F E0   ADD    R0, PC, R0
__text:0000292C 06 30 A0 E3   MOV    R3, #6
__text:00002930 05 10 A0 E3   MOV    R1, #5
__text:00002934 00 20 8D E5   STR    R2, [SP,#0x1C+var_1C]
__text:00002938 0A 10 8D E9   STMFA  SP, {R1,R3,R12}
__text:0000293C 08 90 A0 E3   MOV    R9, #8
__text:00002940 01 10 A0 E3   MOV    R1, #1
__text:00002944 02 20 A0 E3   MOV    R2, #2
__text:00002948 03 30 A0 E3   MOV    R3, #3
__text:0000294C 10 90 8D E5   STR    R9, [SP,#0x1C+var_C]
__text:00002950 A4 05 00 EB   BL     _printf
__text:00002954 07 D0 A0 E1   MOV    SP, R7
__text:00002958 80 80 BD E8   LDMFD  SP!, {R7,PC}
```

Niemal to samo widzieliśmy poprzednio, za wyjątkiem istrukcji STMFA (Store Multiple Full Ascending), która jest synonimem STMIB (Store Multiple Increment Before). Instrukcja najpierw zwiększa wartość rejestru SP, a po tym zapisuje wartości rejestrów (z drugiego operandu) do pamięci. Te dwa kroki odbywają się w odwrotnej kolejności niż w instrukcji STMIA.

Można również zwrócić uwagę, że instrukcje zostały rozrzucone jakby losowo. Na przykład wartość w rejestrze R0 jest ustawiana w trzech różnych miejscach: 0x2918, 0x2920 i 0x2928, a można by to zrobić w jednym.

Musimy pamiętać, że ten porządek jest pozornie losowy, kompilator ma swoje powodu do takiego szeregowania instrukcji, ponieważ kieruje się efektywnością kodu w trakcie wykonania.

Procesor z reguły próbuje równolegle wykonywać instrukcje położone obok siebie. Na przykład, instrukcje jak MOVT R0, #0 oraz ADD R0, PC, R0 nie mogę być wykonywane równocześnie, gdyż obie modyfikują ten sam rejestr R0. Z drugiej strony, MOVT R0, #0 oraz MOV R2, #4 mogą być wykonywane równolegle, gdyż nie ma konfliktu między wynikami ich pracy. Prawdopodobnie kompilator starał się tak uszeregować instrukcje, by mogły być wykonywane równolegle tam, gdzie to możliwe.

Optymalizujący Xcode 4.6.3 (LLVM): tryb Thumb-2

```
__text:00002BA0               _printf_main2
__text:00002BA0
__text:00002BA0               var_1C = -0x1C
__text:00002BA0               var_18 = -0x18
__text:00002BA0               var_C  = -0xC
__text:00002BA0
__text:00002BA0 80 B5          PUSH     {R7,LR}
__text:00002BA2 6F 46          MOV      R7, SP
__text:00002BA4 85 B0          SUB      SP, SP, #0x14
__text:00002BA6 41 F2 D8 20    MOVW     R0, #0x12D8
__text:00002BAA 4F F0 07 0C    MOV.W    R12, #7
__text:00002BAE C0 F2 00 00    MOVT.W   R0, #0
__text:00002BB2 04 22          MOVS     R2, #4
__text:00002BB4 78 44          ADD      R0, PC  ; char *
__text:00002BB6 06 23          MOVS     R3, #6
```

```
__text:00002BB8 05 21          MOVS     R1, #5
__text:00002BBA 0D F1 04 0E    ADD.W    LR, SP, #0x1C+var_18
__text:00002BBE 00 92          STR      R2, [SP,#0x1C+var_1C]
__text:00002BC0 4F F0 08 09    MOV.W    R9, #8
__text:00002BC4 8E E8 0A 10    STMIA.W  LR, {R1,R3,R12}
__text:00002BC8 01 21          MOVS     R1, #1
__text:00002BCA 02 22          MOVS     R2, #2
__text:00002BCC 03 23          MOVS     R3, #3
__text:00002BCE CD F8 10 90    STR.W    R9, [SP,#0x1C+var_C]
__text:00002BD2 01 F0 0A EA    BLX      _printf
__text:00002BD6 05 B0          ADD      SP, SP, #0x14
__text:00002BD8 80 BD          POP      {R7,PC}
```

Wyjście jest prawie takie samo jak w poprzednim przykładzie, różnicą jest użycie instrukcji z trybu Thumb/Thumb-2.

ARM64

Nieoptymalizujący GCC (Linaro) 4.9

Listing 1.57: Nieoptymalizujący GCC (Linaro) 4.9

```
.LC2:
        .string "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n"
f3:
; przydziel miejsce na stosie:
        sub     sp, sp, #32
; zapisz FP i LR w ramce stosu:
        stp     x29, x30, [sp,16]
; ustaw wskaźnik ramki stosu (FP=SP+16):
        add     x29, sp, 16
        adrp    x0, .LC2 ; "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n"
        add     x0, x0, :lo12:.LC2
        mov     w1, 8           ; 9. argument
        str     w1, [sp]        ; zapisz 9. argument na stosie
        mov     w1, 1
        mov     w2, 2
        mov     w3, 3
        mov     w4, 4
        mov     w5, 5
        mov     w6, 6
        mov     w7, 7
        bl      printf
        sub     sp, x29, #16
; przywróć FP i LR
        ldp     x29, x30, [sp,16]
        add     sp, sp, 32
        ret
```

Pierwsze 8 argumentów przekazywanych jest w rejestrach X- oraz W-: [*Procedure Call Standard for the ARM 64-bit Architecture (AArch64)*, (2013)][72]. Wskaźnik na łańcuch znaków wymaga 64-bitowego rejestru, trafia więc do X0. Wszystkie pozostałe wartości są typu *int*, mają szerokość 32 bitów i umieszczane są w młodszych 32-bitowych częściach rejestrów (W-). Dziewiąty argument (8) jest przekazywany przez stos. Nie można przekazać dużej liczby argumentów przez rejestry, gdyż ich liczba jest ograniczona.

Optymalizujący GCC (Linaro) 4.9 generuje taki sam kod.

1.11.3 MIPS

4 argumenty

Optymalizujący GCC 4.4.5

Główną różnicą między przykładem z „Hello, world!” jest to, że tym razem wywoływana jest funkcja `printf()` zamiast `puts()` a 3 dodatkowe argumenty przekazywane są przez rejestry $5…$7 ($A1 …$A3). Nazwy tych rejestrów są poprzedzone literą „A”, gdyż w architekturze MIPS rejestry $4…$7 ($A0 …$A3) służą do przekazywania argumentów.

Listing 1.58: Optymalizujący GCC 4.4.5 (wyjście w asemblerze)

```
$LC0:
        .ascii  "a=%d; b=%d; c=%d\000"
main:
; prolog:
        lui     $28,%hi(__gnu_local_gp)
        addiu   $sp,$sp,-32
        addiu   $28,$28,%lo(__gnu_local_gp)
        sw      $31,28($sp)
; załaduj adres funkcji printf():
        lw      $25,%call16(printf)($28)
; załaduj adres łańcucha znaków i ustaw jako 1. argument funkcji printf():
        lui     $4,%hi($LC0)
        addiu   $4,$4,%lo($LC0)
; ustaw 2. argument funkcji printf():
        li      $5,1                    # 0x1
; ustaw 3. argument funkcji printf():
        li      $6,2                    # 0x2
; wywołaj printf():
        jalr    $25
; ustaw 4. argument funkcji printf() (branch delay slot):
        li      $7,3                    # 0x3

; epilog:
        lw      $31,28($sp)
; ustaw wartość zwracaną na 0:
        move    $2,$0
```

[72] Dostęp także przez http://infocenter.arm.com/help/topic/com.arm.doc.ihi0055b/IHI0055B_aapcs64.pdf

```
; zwróć wartość:
        j       $31
        addiu   $sp,$sp,32 ; branch delay slot
```

Listing 1.59: Optymalizujący GCC 4.4.5 (IDA)

```
.text:00000000 main:
.text:00000000
.text:00000000 var_10          = -0x10
.text:00000000 var_4           = -4
.text:00000000
; prolog:
.text:00000000                 lui     $gp, (__gnu_local_gp >> 16)
.text:00000004                 addiu   $sp, -0x20
.text:00000008                 la      $gp, (__gnu_local_gp & 0xFFFF)
.text:0000000C                 sw      $ra, 0x20+var_4($sp)
.text:00000010                 sw      $gp, 0x20+var_10($sp)
; załaduj adres funkcji printf():
.text:00000014                 lw      $t9, (printf & 0xFFFF)($gp)
; załaduj adres łańcucha znaków i ustaw jako 1. argument funkcji printf():
.text:00000018                 la      $a0, $LC0        # "a=%d; b=%d; c=%d"
; ustaw 2. argument funkcji printf():
.text:00000020                 li      $a1, 1
; ustaw 3. argument funkcji printf():
.text:00000024                 li      $a2, 2
; call printf():
.text:00000028                 jalr    $t9
; ustaw 4. argument funkcji printf() (branch delay slot):
.text:0000002C                 li      $a3, 3
; epilog:
.text:00000030                 lw      $ra, 0x20+var_4($sp)
; ustaw wartość zwracaną na 0:
.text:00000034                 move    $v0, $zero
; zwróć wartość:
.text:00000038                 jr      $ra
.text:0000003C                 addiu   $sp, 0x20 ; branch delay slot
```

IDA zgrupowała parę instrukcji LUI i ADDIU w jedną pseudoinstrukcję LA. Z tego powodu pod adresem 0x1C nie ma instrukcji - LA *zajmuje* 8 bajtów.

Nieoptymalizujący GCC 4.4.5

Nieoptymalizujący GCC generuje nieco rozwlekły kod:

Listing 1.60: Nieoptymalizujący GCC 4.4.5 (wyjście w asemblerze)

```
$LC0:
        .ascii  "a=%d; b=%d; c=%d\000"
main:
; prolog:
        addiu   $sp,$sp,-32
        sw      $31,28($sp)
```

```
        sw      $fp,24($sp)
        move    $fp,$sp
        lui     $28,%hi(__gnu_local_gp)
        addiu   $28,$28,%lo(__gnu_local_gp)
; załaduj adres łańcucha znaków:
        lui     $2,%hi($LC0)
        addiu   $2,$2,%lo($LC0)
; ustaw 1. argument funkcji printf():
        move    $4,$2
; ustaw 2. argument funkcji printf():
        li      $5,1                    # 0x1
; ustaw 3. argument funkcji printf():
        li      $6,2                    # 0x2
; ustaw 4. argument funkcji printf():
        li      $7,3                    # 0x3
; pobierz adres funkcji printf():
        lw      $2,%call16(printf)($28)
        nop
; wywołaj printf():
        move    $25,$2
        jalr    $25
        nop

; epilog:
        lw      $28,16($fp)
; ustaw wartość zwracaną na 0:
        move    $2,$0
        move    $sp,$fp
        lw      $31,28($sp)
        lw      $fp,24($sp)
        addiu   $sp,$sp,32
; zwróć wartość:
        j       $31
        nop
```

Listing 1.61: Nieoptymalizujący GCC 4.4.5 (IDA)

```
.text:00000000 main:
.text:00000000
.text:00000000 var_10          = -0x10
.text:00000000 var_8           = -8
.text:00000000 var_4           = -4
.text:00000000
; prolog:
.text:00000000                 addiu   $sp, -0x20
.text:00000004                 sw      $ra, 0x20+var_4($sp)
.text:00000008                 sw      $fp, 0x20+var_8($sp)
.text:0000000C                 move    $fp, $sp
.text:00000010                 la      $gp, __gnu_local_gp
.text:00000018                 sw      $gp, 0x20+var_10($sp)
; załaduj adres łańcucha znaków:
.text:0000001C                 la      $v0, aADBDCD     # "a=%d; b=%d; c=%d"
; ustaw 1. argument funkcji printf():
.text:00000024                 move    $a0, $v0
```

```
; ustaw 2. argument funkcji printf():
.text:00000028                 li      $a1, 1
; ustaw 3. argument funkcji printf():
.text:0000002C                 li      $a2, 2
; ustaw 4. argument funkcji printf():
.text:00000030                 li      $a3, 3
; pobierz adres funkcji printf():
.text:00000034                 lw      $v0, (printf & 0xFFFF)($gp)
.text:00000038                 or      $at, $zero
; wywołaj printf():
.text:0000003C                 move    $t9, $v0
.text:00000040                 jalr    $t9
.text:00000044                 or      $at, $zero ; NOP
; epilog:
.text:00000048                 lw      $gp, 0x20+var_10($fp)
; ustaw wartość zwracaną na 0:
.text:0000004C                 move    $v0, $zero
.text:00000050                 move    $sp, $fp
.text:00000054                 lw      $ra, 0x20+var_4($sp)
.text:00000058                 lw      $fp, 0x20+var_8($sp)
.text:0000005C                 addiu   $sp, 0x20
; zwróć wartość:
.text:00000060                 jr      $ra
.text:00000064                 or      $at, $zero ; NOP
```

9 argumentów

Ponownie użyjmy przykładu z 9. argumentami z poprzedniego rozdziału: 1.11.1 on page 68.

```
#include <stdio.h>

int main()
{
        printf("a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n", 1, 2, 3,↙
    ↘ 4, 5, 6, 7, 8);
        return 0;
};
```

Optymalizujący GCC 4.4.5

4 pierwsze argumenty są przekazane przez rejestry $A0 …$A3, a pozostałe przez stos.

Jest to tzw. konwencja wywoływania O32 (najpowszechniejsza w świecie MIPS). Inne konwencje (jak np. N32) mogą używać innej liczby rejestrów do przekazywania argumentów.

SW to skrótowiec od „Store Word” (z rejestru do pamięci). W MIPS brakuje instrukcji bezpośrednio zapisującej wartość w pamięci, zawsze do tego celu trzeba użyć pary LI/SW.

Listing 1.62: Optymalizujący GCC 4.4.5 (wyjście w asemblerze)

```
$LC0:
        .ascii  "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\012\000"
main:
; prolog:
        lui     $28,%hi(__gnu_local_gp)
        addiu   $sp,$sp,-56
        addiu   $28,$28,%lo(__gnu_local_gp)
        sw      $31,52($sp)
; zapisz 5. argument na stosie:
        li      $2,4                    # 0x4
        sw      $2,16($sp)
; zapisz 6. argument na stosie:
        li      $2,5                    # 0x5
        sw      $2,20($sp)
; zapisz 7. argument na stosie:
        li      $2,6                    # 0x6
        sw      $2,24($sp)
; zapisz 8. argument na stosie:
        li      $2,7                    # 0x7
        lw      $25,%call16(printf)($28)
        sw      $2,28($sp)
; zapisz 1 argument w $a0:
        lui     $4,%hi($LC0)
; zapisz 9 argument na stosie:
        li      $2,8                    # 0x8
        sw      $2,32($sp)
        addiu   $4,$4,%lo($LC0)
; zapisz 2 argument w $a1:
        li      $5,1                    # 0x1
; zapisz 3 argument w $a2:
        li      $6,2                    # 0x2
; wywołaj printf():
        jalr    $25
; zapisz 4 argument w $a3 (branch delay slot):
        li      $7,3                    # 0x3

; epilog:
        lw      $31,52($sp)
; ustaw wartość zwracaną na 0:
        move    $2,$0
; zwróć wartość
        j       $31
        addiu   $sp,$sp,56 ; branch delay slot
```

Listing 1.63: Optymalizujący GCC 4.4.5 (IDA)

```
.text:00000000 main:
.text:00000000
.text:00000000 var_28          = -0x28
.text:00000000 var_24          = -0x24
.text:00000000 var_20          = -0x20
.text:00000000 var_1C          = -0x1C
```

```
.text:00000000 var_18          = -0x18
.text:00000000 var_10          = -0x10
.text:00000000 var_4           = -4
.text:00000000
; prolog:
.text:00000000                 lui     $gp, (__gnu_local_gp >> 16)
.text:00000004                 addiu   $sp, -0x38
.text:00000008                 la      $gp, (__gnu_local_gp & 0xFFFF)
.text:0000000C                 sw      $ra, 0x38+var_4($sp)
.text:00000010                 sw      $gp, 0x38+var_10($sp)
; zapisz 5. argument na stosie:
.text:00000014                 li      $v0, 4
.text:00000018                 sw      $v0, 0x38+var_28($sp)
; zapisz 6. argument na stosie:
.text:0000001C                 li      $v0, 5
.text:00000020                 sw      $v0, 0x38+var_24($sp)
; zapisz 7. argument na stosie:
.text:00000024                 li      $v0, 6
.text:00000028                 sw      $v0, 0x38+var_20($sp)
; zapisz 8. argument na stosie:
.text:0000002C                 li      $v0, 7
.text:00000030                 lw      $t9, (printf & 0xFFFF)($gp)
.text:00000034                 sw      $v0, 0x38+var_1C($sp)
; przygotuj 1. argument w $a0:
.text:00000038                 lui     $a0, ($LC0 >> 16)  # "a=%d; b=%d;
    c=%d; d=%d; e=%d; f=%d; g=%"...
; zapisz 9. argument na stosie:
.text:0000003C                 li      $v0, 8
.text:00000040                 sw      $v0, 0x38+var_18($sp)
; zapisz 1. argument w $a0:
.text:00000044                 la      $a0, ($LC0 & 0xFFFF)  # "a=%d; b=%d;
    c=%d; d=%d; e=%d; f=%d; g=%"...
; zapisz 2. argument w $a1:
.text:00000048                 li      $a1, 1
; zapisz 3. argument w $a2:
.text:0000004C                 li      $a2, 2
; wywołaj printf():
.text:00000050                 jalr    $t9
; zapisz 4. argument w $a3 (branch delay slot):
.text:00000054                 li      $a3, 3
; epilog:
.text:00000058                 lw      $ra, 0x38+var_4($sp)
; ustaw wartość zwracaną na 0:
.text:0000005C                 move    $v0, $zero
; zwróć wartość
.text:00000060                 jr      $ra
.text:00000064                 addiu   $sp, 0x38 ; branch delay slot
```

Nieoptymalizujący GCC 4.4.5

Nieoptymalizujący GCC generuje nieco rozwlekły kod:

Listing 1.64: Nieoptymalizujący GCC 4.4.5 (wyjście w asemblerze)

```
$LC0:
        .ascii  "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\012\000"
main:
; prolog:
        addiu   $sp,$sp,-56
        sw      $31,52($sp)
        sw      $fp,48($sp)
        move    $fp,$sp
        lui     $28,%hi(__gnu_local_gp)
        addiu   $28,$28,%lo(__gnu_local_gp)
        lui     $2,%hi($LC0)
        addiu   $2,$2,%lo($LC0)
; zapisz 5. argument na stosie:
        li      $3,4                    # 0x4
        sw      $3,16($sp)
; zapisz 6. argument na stosie:
        li      $3,5                    # 0x5
        sw      $3,20($sp)
; zapisz 7. argument na stosie:
        li      $3,6                    # 0x6
        sw      $3,24($sp)
; zapisz 8. argument na stosie:
        li      $3,7                    # 0x7
        sw      $3,28($sp)
; zapisz 9. argument na stosie:
        li      $3,8                    # 0x8
        sw      $3,32($sp)
; zapisz 1. argument w $a0:
        move    $4,$2
; zapisz 2. argument w $a1:
        li      $5,1                    # 0x1
; zapisz 3. argument w $a2:
        li      $6,2                    # 0x2
; zapisz 4. argument w $a3:
        li      $7,3                    # 0x3
; wywołaj printf():
        lw      $2,%call16(printf)($28)
        nop
        move    $25,$2
        jalr    $25
        nop
; epilog:
        lw      $28,40($fp)
; ustaw wartość zwracaną na 0:
        move    $2,$0
        move    $sp,$fp
        lw      $31,52($sp)
        lw      $fp,48($sp)
        addiu   $sp,$sp,56
; zwróć wartość:
        j       $31
        nop
```

Listing 1.65: Nieoptymalizujący GCC 4.4.5 (IDA)

```
.text:00000000 main:
.text:00000000
.text:00000000 var_28          = -0x28
.text:00000000 var_24          = -0x24
.text:00000000 var_20          = -0x20
.text:00000000 var_1C          = -0x1C
.text:00000000 var_18          = -0x18
.text:00000000 var_10          = -0x10
.text:00000000 var_8           = -8
.text:00000000 var_4           = -4
.text:00000000
; prolog:
.text:00000000                 addiu   $sp, -0x38
.text:00000004                 sw      $ra, 0x38+var_4($sp)
.text:00000008                 sw      $fp, 0x38+var_8($sp)
.text:0000000C                 move    $fp, $sp
.text:00000010                 la      $gp, __gnu_local_gp
.text:00000018                 sw      $gp, 0x38+var_10($sp)
.text:0000001C                 la      $v0, aADBDCDDDEDFDGD  # "a=%d; b=%d;
    c=%d; d=%d; e=%d; f=%d; g=%"...
; zapisz 5. argument na stosie:
.text:00000024                 li      $v1, 4
.text:00000028                 sw      $v1, 0x38+var_28($sp)
; zapisz 6. argument na stosie:
.text:0000002C                 li      $v1, 5
.text:00000030                 sw      $v1, 0x38+var_24($sp)
; zapisz 7. argument na stosie:
.text:00000034                 li      $v1, 6
.text:00000038                 sw      $v1, 0x38+var_20($sp)
; zapisz 8. argument na stosie:
.text:0000003C                 li      $v1, 7
.text:00000040                 sw      $v1, 0x38+var_1C($sp)
; zapisz 9. argument na stosie:
.text:00000044                 li      $v1, 8
.text:00000048                 sw      $v1, 0x38+var_18($sp)
; zapisz 1. argument w $a0:
.text:0000004C                 move    $a0, $v0
; zapisz 2. argument w $a1:
.text:00000050                 li      $a1, 1
; zapisz 3. argument w $a2:
.text:00000054                 li      $a2, 2
; zapisz 4. argument w $a3:
.text:00000058                 li      $a3, 3
; wywołaj printf():
.text:0000005C                 lw      $v0, (printf & 0xFFFF)($gp)
.text:00000060                 or      $at, $zero
.text:00000064                 move    $t9, $v0
.text:00000068                 jalr    $t9
.text:0000006C                 or      $at, $zero ; NOP
; epilog:
.text:00000070                 lw      $gp, 0x38+var_10($fp)
; ustaw wartość zwracaną na 0:
```

```
.text:00000074                 move    $v0, $zero
.text:00000078                 move    $sp, $fp
.text:0000007C                 lw      $ra, 0x38+var_4($sp)
.text:00000080                 lw      $fp, 0x38+var_8($sp)
.text:00000084                 addiu   $sp, 0x38
; zwróć wartość:
.text:00000088                 jr      $ra
.text:0000008C                 or      $at, $zero ; NOP
```

1.11.4 Wnioski

Tak wygląda szkielet wywołania funkcji:

Listing 1.66: x86

```
...
PUSH 3. argument
PUSH 2. argument
PUSH 1. argument
CALL funkcja
; zmodyfikuj wskaźnik stosu (jeśli trzeba)
```

Listing 1.67: x64 (MSVC)

```
MOV RCX, 1. argument
MOV RDX, 2. argument
MOV R8, 3. argument
MOV R9, 4. argument
...
PUSH 5., 6., ... argument ; (jeśli trzeba)
CALL funkcja
; zmodyfikuj wskaźnik stosu (jeśli trzeba)
```

Listing 1.68: x64 (GCC)

```
MOV RDI, 1. argument
MOV RSI, 2. argument
MOV RDX, 3. argument
MOV RCX, 4. argument
MOV R8, 5. argument
MOV R9, 6. argument
...
PUSH 7., 8., ... argument ; (jeśli trzeba)
CALL funkcja
; zmodyfikuj wskaźnik stosu (jeśli trzeba)
```

Listing 1.69: ARM

```
MOV R0, 1. argument
MOV R1, 2. argument
MOV R2, 3. argument
MOV R3, 4. argument
; przekaż 5., 6., ... argument przez stos (jeśli trzeba)
```

```
BL funkcja
; zmodyfikuj wskaźnik stosu (jeśli trzeba)
```

Listing 1.70: ARM64

```
MOV X0, 1. argument
MOV X1, 2. argument
MOV X2, 3. argument
MOV X3, 4. argument
MOV X4, 5. argument
MOV X5, 6. argument
MOV X6, 7. argument
MOV X7, 8. argument
; przekaż 9., 10., ... argument przez stos (jeśli trzeba)
BL funkcja
; zmodyfikuj wskaźnik stosu (jeśli trzeba)
```

Listing 1.71: MIPS (konwencja wywoływania O32)

```
LI $4, 1. argument ; AKA $A0
LI $5, 2. argument ; AKA $A1
LI $6, 3. argument ; AKA $A2
LI $7, 4. argument ; AKA $A3
; przekaż 5., 6., ... argument, przez stos (jeśli trzeba)
LW temp_reg, adress funkcji
JALR temp_reg
```

1.11.5 Przy okazji

Różnice między przekazywaniem argumentów funkcji w x86, x64, fastcall, ARM i MIPS pokazują, że procesorowi jest bez różnicy, jak będą przekazywane argumenty do funkcji. Można by stworzyć kompilator, który będzie je przekazywał za pomocą wskaźnika na strukturę z argumentami, nie korzystając ze stosu w ogóle.

Rejestry \$A0…\$A3 w MIPS są nazwane w ten sposób tylko dla wygody (jest tak w konwencji wywoływania O32). Programiści mogą korzystać z jakichkolwiek innych rejestrów (może oprócz \$ZERO) do przekazywania argumentów i dowolnej konwencji wywoływania funkcji.

CPU w żaden sposób nie jest świadomy jakiej metody używamy.

Początkujący programiści asemblera często przekazują argumenty do funkcji przez rejestry, bez wyraźnego porządku lub nawet przez zmienne globalne.

To też będzie działać poprawnie.

1.12 scanf()

Tym razem zajmiemy się funkcją `scanf()`

1.12.1 Prosty przykład

```
#include <stdio.h>

int main()
{
        int x;
        printf ("Enter X:\n");

        scanf ("%d", &x);

        printf ("You entered %d...\n", x);

        return 0;
};
```

Używanie `scanf()` do interakcji z użytkownikiem nie jest dobrym pomysłem w dzisiejszych czasach, jednak mimo wszystko funkcja ta jest dobrym przykładem użycia wskaźnika na zmienną typu *int*.

O wskaźnikach

Wskaźniki są jednym z podstawowych pojęć informatycznych. Często przekazywanie dużej tablicy, struktury lub obiektu do innej funkcji jest pamięciożerne, podczas gdy przekazanie samego adresu jest znacznie tańsze.

Kiedy chcesz wypisać tekst w konsoli, najprościej będzie wskazać jego adres w pamięci.

W dodatku, jeśli wywoływana funkcja potrzebuje zmodyfikować cokolwiek w dużej tablicy lub strukturze danych przekazanej jako parametr a następnie ją zwrócić, kopiowanie tylu danych byłoby prawie absurdalne. Dlatego najprościej będzie przekazać adres tej tablicy/struktury do wywoływanej funkcji i wtedy zmodyfikować to, co wymaga modyfikacji.

Wskaźnik w C/C++—jest adresem pewnego miejsca w pamięci.

W x86 adresy są reprezentowane przy pomocy 32-bitowych liczb (czyli 4 bajtowych), a w x86-64 jako liczby 64-bitowe (czyli 8 bajtowe). Przy okazji jest to powód dlaczego niektórych ludzi oburza przeskok na x86-64—wszystkie wskaźniki w architekturze x64 wymagają dwa razy więcej miejsca, włączając pamięć cache, która jest bardzo ”kosztowna”.

Można pracować jedynie z nietypowanymi wskaźnikami, wymaga to jednak nieco wysiłku, np. użycia funkcji z biblioteki standardowej C `memcpy()`, która kopiuje blok z jednego miejsca w pamięci do drugiego. `memcpy()` jako argumenty przyjmuje 2 wskaźniki typu `void*`, co umożliwia kopiowanie dowolnych typów danych. Typy danych nie są istotne, znaczenie mają tylko rozmiary bloków pamięci.

Wskaźniki są także często używane kiedy funkcja potrzebuje zwrócić więcej niż jedną wartość (wrócę do tego później (**??** on page ??)).

Funkcja *scanf()* jest takim przypadkiem. Poza tym, że funkcja `scanf()` zwraca liczbę wczytanych wartości, to musi jeszcze je jakoś przekazać.

W C/C++ typ wskaźnika jest potrzebny tylko do sprawdzania typów podczas kompilacji.

W skompilowanym kodzie nie ma żadnej informacji jakiego typu są wskaźniki.

x86

MSVC

Poniższy kod otrzymamy po kompilacji za pomocą MSVC 2010:

```
CONST    SEGMENT
$SG3831    DB    'Enter X:', 0aH, 00H
$SG3832    DB    '%d', 00H
$SG3833    DB    'You entered %d...', 0aH, 00H
CONST    ENDS
PUBLIC    _main
EXTRN    _scanf:PROC
EXTRN    _printf:PROC
; Ustawione opcje kompilacji funkcji: /Odtp
_TEXT    SEGMENT
_x$ = -4                        ; size = 4
_main    PROC
    push   ebp
    mov    ebp, esp
    push   ecx
    push   OFFSET $SG3831 ; 'Enter X:'
    call   _printf
    add    esp, 4
    lea    eax, DWORD PTR _x$[ebp]
    push   eax
    push   OFFSET $SG3832 ; '%d'
    call   _scanf
    add    esp, 8
    mov    ecx, DWORD PTR _x$[ebp]
    push   ecx
    push   OFFSET $SG3833 ; 'You entered %d...'
    call   _printf
    add    esp, 8

    ; zwróć 0
    xor    eax, eax
    mov    esp, ebp
    pop    ebp
    ret    0
_main    ENDP
_TEXT    ENDS
```

x jest zmienną lokalną.

Według standardu C/C++ zmienna lokalna może być widoczna tylko w konkretnej funkcji. Tradycyjnie zmienne lokalne są przechowywane na stosie. Prawdopodobnie są inne moliwości przechowywania tych zmiennych, ale tak akurat jest w x86.

Zadaniem instrukcji rozpoczynającej funkcję, PUSH ECX, nie jest zapisanie stanu ECX (można zauważyć brak odpowiadającej instrukcji POP ECX na końcu funkcji).

Tak naprawdę instrukcja ta alokuje 4 bajty na stosie do przechowania zmiennej x.

Dostęp do x odbywa się za pomocą makra _x$ (-4) i rejestru EBP, który wskazuje na bieżącą ramkę.

W trakcie wykonywania funkcji EBP wskazuje na bieżącą ramkę stosu, umożliwiając dostęp do zmiennych lokalnych i argumentów funkcji poprzez EBP+`offset`.

Można by użyć w tym celu rejestru ESP, ale nie byłoby to zbyt wygodne, ponieważ wartość tego rejestru często się zmienia. Wartość EBP może być postrzegana jako *zachowana* wartość ESP z początku wykonania funkcji.

Poniżej pokazano typowy układ ramki stosu w środowisku 32-bitowym:

…	…
EBP-8	zmienna lokalna #2, oznaczony w programie IDA jako `var_8`
EBP-4	zmienna lokalna #1, oznaczony w programie IDA jako `var_4`
EBP	zapisana wartość EBP
EBP+4	adres powrotu
EBP+8	argument#1, oznaczony w programie IDA jako `arg_0`
EBP+0xC	argument#2, oznaczony w programie IDA jako `arg_4`
EBP+0x10	argument#3, oznaczony w programie IDA jako `arg_8`
…	…

Funkcja `scanf()` w naszym przykładzie ma dwa argumenty.

Pierwszy jest wskaźnikiem na łańcuch znaków %d a drugi jest adresem zmiennej x.

Na początku adres zmiennej x jest ładowany do rejestru EAX przy pomocy instrukcji `lea eax, DWORD PTR _x$[ebp]`.

LEA oznacza *load effective address* i jest często używana do formowania adresów (**??** on page ??).

Można powiedzieć, że w tym przypadku LEA po prostu umieszcza sumę rejestru EBP i makra _x$ w rejestrze EAX.

W tym przypadku (_x$ = -4) jest to samo co `lea eax, [ebp-4]`. Więc od rejestru EBP jest odejmowane 4 i wynik zostaje umieszczony w rejestrze EAX. Następnie wartość rejestru EAX jest odkładana na stos i funkcja `scanf()` zostaje wywołana.

Kolejno następuje przygotowanie do wywołania funkcji `printf()`. Pierwszym argumentem jest wskaźnik na łańcuch znaków: `You entered %d...\n`.

Drugi argument jest przygotowywany za pomocą: `mov ecx, [ebp-4]`. Instrukcja kopiuje zmienną x (nie jej adres) do rejestru ECX.

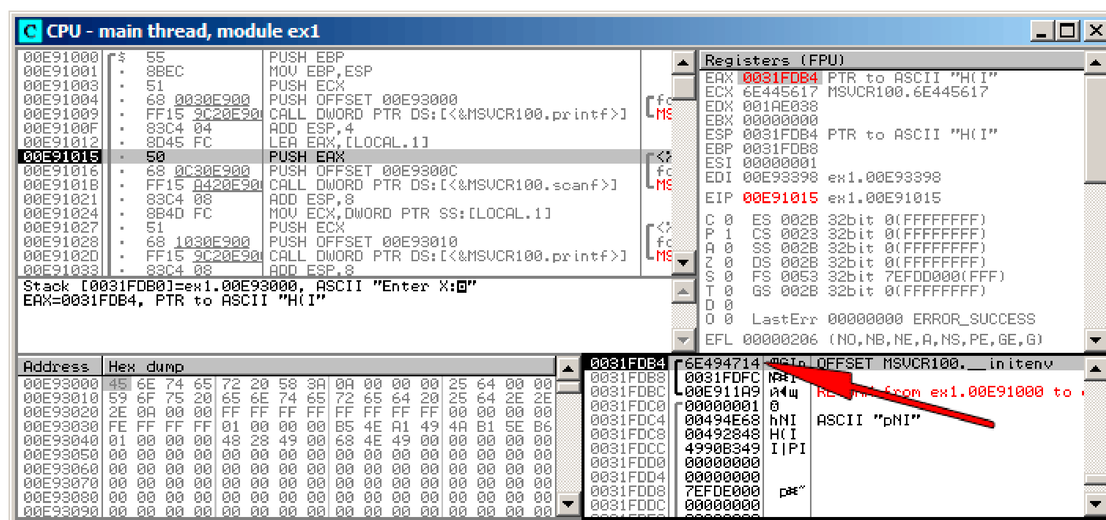
Następnie wartość z ECX jest odkładana na stos, a na koniec zostaje wywołana funkcja `printf()`.

MSVC + OllyDbg

Otwórzmy przykład w OllyDbg. Po załadowaniu wciskamy kilka razy F8, aż dotrzemy do naszego pliku wykonywalnego, zamiast `ntdll.dll`. Scrollujemy na górę, aż pojawi się funkcja `main()`.

Kliknij na pierwszą instrukcję (PUSH EBP) i naciśnij F2 (*ustaw breakpoint*), a następnie F9 (*Run*). Zatrzymamy się na początku funkcji main.

Przejdźmy do miejsca, w którym wyliczany jest adres zmiennej x:



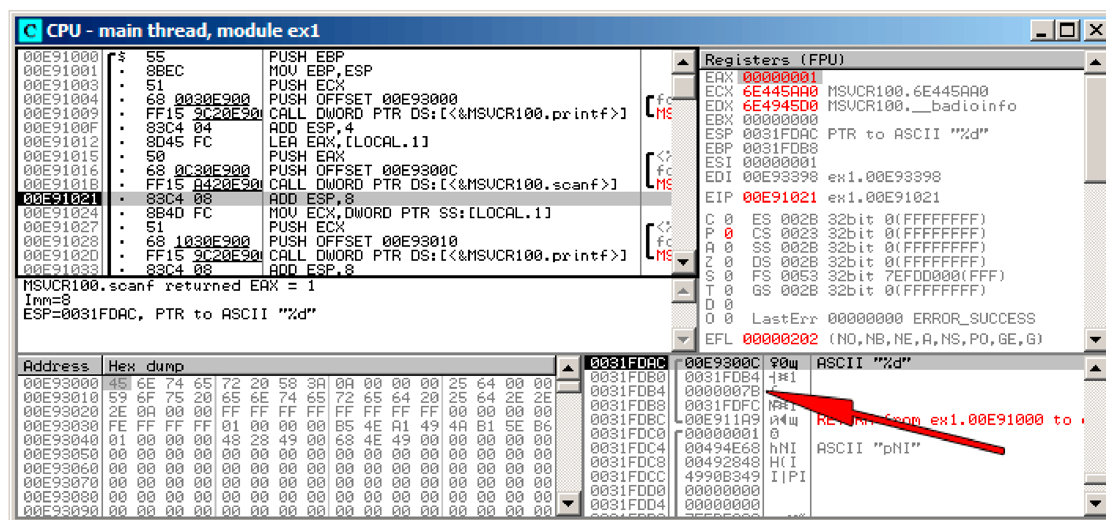
Rysunek 1.13: OllyDbg: wyliczanie adresu zmiennej lokalnej

Kliknij prawym przyciskiem na rejestr EAX w oknie z rejestrami i wybierz „Follow in stack”.

Adres z EAX pojawi się w oknie z widokiem stosu. Czerwona strzałka pokazuje na zmienną lokalną na stosie. W tej chwili są tam śmieci — (0x6E494714). Za pomocą instrukcji PUSH adres tego elementu na stosie również trafi na stos, jako kolejny element. Wciskając F8, przejdźmy za wywołanie funkcji `scanf()`. W trakcie wykonywania funkcji musimy podać jakiś wejściowy ciąg znaków w oknie konsoli, np. „123”.

```
Enter X:
123
```

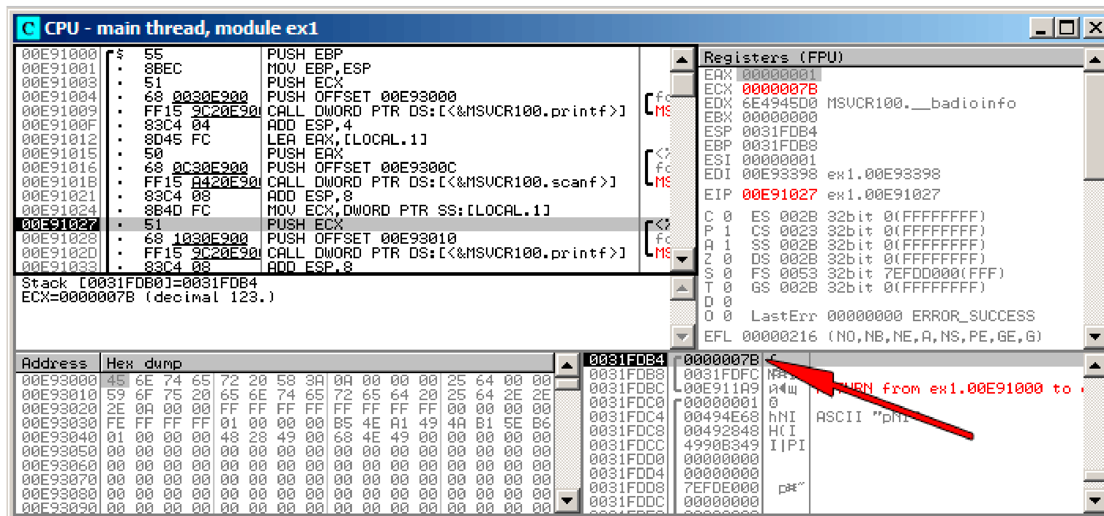
Funkcja scanf() zakończyła swoje wykonanie.



Rysunek 1.14: OllyDbg: stan po zakończeniu funkcji scanf()

Funkcja scanf() zwróciła 1 w EAX, co oznacza, że wczytała jedną wartość. Jeśli ponownie spojrzymy na element na stosie odpowiadający zmiennej lokalnej, zobaczymy, że ma on teraz wartość 0x7B (123).

Później wartość zostanie skopiowana ze stosu do rejestru ECX i przekazana do funkcji `printf()`:



Rysunek 1.15: OllyDbg: przygotowanie argumentu funkcji `printf()`

GCC

Tak wygląda skompilowany kod w GCC 4.4.1 w systemie Linux:

```
main            proc near

var_20          = dword ptr -20h
var_1C          = dword ptr -1Ch
var_4           = dword ptr -4

                push    ebp
                mov     ebp, esp
                and     esp, 0FFFFFFF0h
                sub     esp, 20h
                mov     [esp+20h+var_20], offset aEnterX ; "Enter X:"
                call    _puts
                mov     eax, offset aD  ; "%d"
                lea     edx, [esp+20h+var_4]
                mov     [esp+20h+var_1C], edx
                mov     [esp+20h+var_20], eax
                call    ___isoc99_scanf
                mov     edx, [esp+20h+var_4]
                mov     eax, offset aYouEnteredD___ ; "You entered %d...\n"
                mov     [esp+20h+var_1C], edx
                mov     [esp+20h+var_20], eax
                call    _printf
                mov     eax, 0
```

```
                leave
                retn
main            endp
```

GCC zamienia wywołanie funkcji `printf()` na wywołanie funkcji `puts()`. Powód tego został wyjaśniony w (1.5.3 on page 29).

Jak w przykładzie z MSVC—argumenty funkcji są umieszczane na stosie przy użyciu instrukcji `MOV`.

Nawiasem mówiąc…

Ten prosty przykład pokazuje, że kompilatory rzeczywiście tłumaczą listę instrukcji języka C na serię instrukcji kodu maszynowego. W kodzie C wykonanie odbywa się instrukcja po instrukcji i podobnie jest w kodzie maszynowym - między instrukcjami nie ma nic więcej.

x64

Sposób wykonania programy będzie niemal taki sam, z tą różnicą, że argumenty tym razem będą będą przekazywane przez rejestry, a nie przez stos.

MSVC

Listing 1.72: MSVC 2012 x64

```
_DATA   SEGMENT
$SG1289 DB      'Enter X:', 0aH, 00H
$SG1291 DB      '%d', 00H
$SG1292 DB      'You entered %d...', 0aH, 00H
_DATA   ENDS

_TEXT   SEGMENT
x$ = 32
main    PROC
$LN3:
        sub     rsp, 56
        lea     rcx, OFFSET FLAT:$SG1289 ; 'Enter X:'
        call    printf
        lea     rdx, QWORD PTR x$[rsp]
        lea     rcx, OFFSET FLAT:$SG1291 ; '%d'
        call    scanf
        mov     edx, DWORD PTR x$[rsp]
        lea     rcx, OFFSET FLAT:$SG1292 ; 'You entered %d...'
        call    printf

        ; zwróć 0
        xor     eax, eax
        add     rsp, 56
        ret     0
```

```
main    ENDP
_TEXT   ENDS
```

GCC

Listing 1.73: Optymalizujący GCC 4.4.6 x64

```
.LC0:
        .string "Enter X:"
.LC1:
        .string "%d"
.LC2:
        .string "You entered %d...\n"

main:
        sub     rsp, 24
        mov     edi, OFFSET FLAT:.LC0 ; "Enter X:"
        call    puts
        lea     rsi, [rsp+12]
        mov     edi, OFFSET FLAT:.LC1 ; "%d"
        xor     eax, eax
        call    __isoc99_scanf
        mov     esi, DWORD PTR [rsp+12]
        mov     edi, OFFSET FLAT:.LC2 ; "You entered %d...\n"
        xor     eax, eax
        call    printf

        ; zwróć 0
        xor     eax, eax
        add     rsp, 24
        ret
```

ARM

Optymalizujący Keil 6/2013 (tryb Thumb)

```
.text:00000042             scanf_main
.text:00000042
.text:00000042             var_8          = -8
.text:00000042
.text:00000042 08 B5        PUSH    {R3,LR}
.text:00000044 A9 A0        ADR     R0, aEnterX  ; "Enter X:\n"
.text:00000046 06 F0 D3 F8  BL      __2printf
.text:0000004A 69 46        MOV     R1, SP
.text:0000004C AA A0        ADR     R0, aD  ; "%d"
.text:0000004E 06 F0 CD F8  BL      __0scanf
.text:00000052 00 99        LDR     R1, [SP,#8+var_8]
.text:00000054 A9 A0        ADR     R0, aYouEnteredD___ ; "You entered
    %d...\n"
.text:00000056 06 F0 CB F8  BL      __2printf
```

```
.text:0000005A 00 20        MOVS    R0, #0
.text:0000005C 08 BD        POP     {R3,PC}
```

Funkcja `scanf()` potrzebuje argumentu— wskaźnika na *int*, by mogła zapisać wyczytaną wartość. *int* jest 32-bitowy i zmieści się idealnie do 32-bitowego rejestru. Miejsce na zmienną lokalną x jest zaalokowane na stosie, IDA oznaczyła przesunięcie względem SP makrem *var_8*. Można by się bez niego obyć, gdyż SP (wskaźnik stosu) już pokazuje na to miejsce i mógły być użyty bezpośrednio.

Wartość z SP jest kopiowany do rejestru R1 i razem z łańcuchem znaków formatu przekazywana jako argumenty do funkcji `scanf()`.

Instrukcja PUSH/POP zachowuje się inaczej niż na x86 (odwrotnie). Są synonimami instrukcji
STM/STMDB/LDM/LDMIA. PUSH najpierw zapisuje wartość na stosie, *a następnie* zmniejsza SP o 4. POP najpierw dodaje 4 do SP, *a następnie* wczytuje wartość ze stosu. Stąd po wykonaniu PUSH, SP pokazuje na nieużywane miejsce na stosie. Zostanie ono wykorzystane przez `scanf()`, a następnie `printf()` do zapisania i wczytania zmiennej lokalnej.

LDMIA oznacza *Load Multiple Registers Increment address After each transfer*. STMDB oznacza *Store Multiple Registers Decrement address Before each transfer*.

Później, za pomocą instrukcji LDR, wartość zmiennej lokalnej jest wczytywana ze stosu do rejestru R1, by następnie zostać przekazana do funkcji `printf()`.

ARM64

Listing 1.74: Nieoptymalizujący GCC 4.9.1 ARM64

```
.LC0:
        .string "Enter X:"
.LC1:
        .string "%d"
.LC2:
        .string "You entered %d...\n"
scanf_main:
; odejmij 32 od SP, a następnie zapisz FP i LR w ramce stosu
        stp     x29, x30, [sp, -32]!
; ustaw wskaźnik ramki stosu (FP=SP)
        add     x29, sp, 0
; wczytaj wskaźnik na łańcuch znaków "Enter X:":
        adrp    x0, .LC0
        add     x0, x0, :lo12:.LC0
; X0=wskaźnik na łańcuch znaków "Enter X:"
; wypisz go:
        bl      puts
; wczytaj wskaźnik na łańcuch znaków "%d":
        adrp    x0, .LC1
        add     x0, x0, :lo12:.LC1
; znajdź miejsce w ramce stosu na zmienną "x" (X1=FP+28):
        add     x1, x29, 28
```

```
23 ; X1=adres zmiennej "x"
24 ; przekaż adres do funkcji scanf() i ją wywołaj:
25        bl      __isoc99_scanf
26 ; załaduj 32-bitową wartość zmiennej z ramki stosu:
27        ldr     w1, [x29,28]
28 ; W1=x
29 ; wczytaj wskaźnik na łańcuch znaków "You entered %d...\n"
30 ; argumenty printf() - łańcuch znaków z X0 i zmienna "x" z X1 (W1 to młodsze
     32 bity)
31        adrp    x0, .LC2
32        add     x0, x0, :lo12:.LC2
33        bl      printf
34 ; zwróć 0
35        mov     w0, 0
36 ; przywróć FP i LR, a następnie dodaj 32 do SP:
37        ldp     x29, x30, [sp], 32
38        ret
```

Na ramkę stosu zaalokowano 32 bajty, a więc więcej niż to konieczne. Być może jest to efekty wyrównywania pamięci? Najciekawszym fragmetem jest szukanie położenia zmiennej x w obrębie ramki stosu (linia 22). Dlaczego 28? Z jakiegoś powodu kompilator zdecydował umieścić zmienną na końcu ramki stosu, a nie na początku. Adres jest przekazywany do funkcji `scanf()`, która umieszcza pod tym adresem wartość wpisaną przez użytkownika. Jest to 32-bitowa wartość typu *int*. Wartość jest pobierana w linii 27 a następnie przekazywana do funkcji `printf()`.

MIPS

Na stosie lokalnym zaalokowano miejsce dla zmiennej x, odwoływać będziemy się do niej przez $\$sp + 24$.

Adres zmiennej przekazywany jest do funkcji `scanf()`. Wartość wpisana przez użytkownika i odczytana za pomocą `scanf()` jest następnie wczytywana za pomocą instrukcji LW („Load Word”) i przekazywana do `printf()`.

Listing 1.75: Optymalizujący GCC 4.4.5 (wyjście w asemblerze)

```
$LC0:
        .ascii  "Enter X:\000"
$LC1:
        .ascii  "%d\000"
$LC2:
        .ascii  "You entered %d...\012\000"
main:
; prolog:
        lui     $28,%hi(__gnu_local_gp)
        addiu   $sp,$sp,-40
        addiu   $28,$28,%lo(__gnu_local_gp)
        sw      $31,36($sp)
; wywołaj puts():
        lw      $25,%call16(puts)($28)
        lui     $4,%hi($LC0)
        jalr    $25
```

```
        addiu   $4,$4,%lo($LC0) ; branch delay slot
; wywołaj scanf():
        lw      $28,16($sp)
        lui     $4,%hi($LC1)
        lw      $25,%call16(__isoc99_scanf)($28)
; ustaw 2.argument funkcji scanf(), $a1=$sp+24:
        addiu   $5,$sp,24
        jalr    $25
        addiu   $4,$4,%lo($LC1) ; branch delay slot

; wywołaj printf():
        lw      $28,16($sp)
; ustaw 2. argument funkcji printf(),
; załaduj słowo word z adresu $sp+24:
        lw      $5,24($sp)
        lw      $25,%call16(printf)($28)
        lui     $4,%hi($LC2)
        jalr    $25
        addiu   $4,$4,%lo($LC2) ; branch delay slot

; epilog:
        lw      $31,36($sp)
; ustaw wartość zwracaną na 0:
        move    $2,$0
; zwróć wartość:
        j       $31
        addiu   $sp,$sp,40      ; branch delay slot
```

IDA wyświetla układ stosu następująco:

Listing 1.76: Optymalizujący GCC 4.4.5 (IDA)

```
.text:00000000 main:
.text:00000000
.text:00000000 var_18   = -0x18
.text:00000000 var_10   = -0x10
.text:00000000 var_4    = -4
.text:00000000
; prolog:
.text:00000000          lui     $gp, (__gnu_local_gp >> 16)
.text:00000004          addiu   $sp, -0x28
.text:00000008          la      $gp, (__gnu_local_gp & 0xFFFF)
.text:0000000C          sw      $ra, 0x28+var_4($sp)
.text:00000010          sw      $gp, 0x28+var_18($sp)
; wywołaj puts():
.text:00000014          lw      $t9, (puts & 0xFFFF)($gp)
.text:00000018          lui     $a0, ($LC0 >> 16)  # "Enter X:"
.text:0000001C          jalr    $t9
.text:00000020          la      $a0, ($LC0 & 0xFFFF)  # "Enter X:" ; branch
    delay slot
; wywołaj scanf():
.text:00000024          lw      $gp, 0x28+var_18($sp)
.text:00000028          lui     $a0, ($LC1 >> 16)  # "%d"
.text:0000002C          lw      $t9, (__isoc99_scanf & 0xFFFF)($gp)
```

```
; ustaw 2. argument funkcji scanf(), $a1=$sp+24:
.text:00000030          addiu   $a1, $sp, 0x28+var_10
.text:00000034          jalr    $t9  ; branch delay slot
.text:00000038          la      $a0, ($LC1 & 0xFFFF)  # "%d"
; call printf():
.text:0000003C          lw      $gp, 0x28+var_18($sp)
; ustaw 2. argument funkcji printf(),
; załaduj słowo z adresu $sp+24:
.text:00000040          lw      $a1, 0x28+var_10($sp)
.text:00000044          lw      $t9, (printf & 0xFFFF)($gp)
.text:00000048          lui     $a0, ($LC2 >> 16)  # "You entered %d...\n"
.text:0000004C          jalr    $t9
.text:00000050          la      $a0, ($LC2 & 0xFFFF)  # "You entered %d...\n"
    ; branch delay slot
; epilog:
.text:00000054          lw      $ra, 0x28+var_4($sp)
; ustaw wartość zwracaną na 0:
.text:00000058          move    $v0, $zero
; zwróć wartość:
.text:0000005C          jr      $ra
.text:00000060          addiu   $sp, 0x28 ; branch delay slot
```

1.12.2 Popularny błąd

Bardzo popularnym błędem jest podanie jako argumentu zmiennej *x* zamiast wskaźnika na zmienną *x*:

```
#include <stdio.h>

int main()
{
        int x;
        printf ("Enter X:\n");

        scanf ("%d", x); // BUG

        printf ("You entered %d...\n", x);

        return 0;
};
```

Co się wtedy stanie? Zmienna *x* jest niezainicjalizowana i zawiera losowe śmieci z lokalnego stosu. Kiedy funkcja `scanf()` jest wywoływana, pobiera ciąg znaków od użytkownika, parsuje jako liczbę i próbuje ją zapisać w *x*, traktując wartość *x* jak adres w pamięci. Jednak skoro tam są losowe śmieci ze stosu, `scanf()` będzie próbować uzyskać dostęp do losowego adresu. Najprawdopodobniej proces natychmiast zakończy się błędem.

Co ciekawe, niektóre biblioteki CRT w wersji do debugowania, umieszczają w pamięci, która jest alokowana, widoczne wzorce takie jak 0xCCCCCCCC albo 0x0BADF00D itp. W tym przypadku *x* może zawierać 0xCCCCCCCC, więc `scanf()` będzie próbowała zapisać pod adres 0xCCCCCCCC. Jeśli zauważysz, że coś w procesie próbuje

zapisać pod 0xCCCCCCCC, to znaczy, że została użyta niezainicjalizowana zmienna (lub wskaźnik). Jest to lepsze rozwiązanie niż gdyby nowo alokowana pamięć była po prostu wyzerowana.

1.12.3 Zmienne globalne

A jeśli zmienna x z poprzedniego przykładu nie będzie zmienną lokalną, a globalną? Wtedy będzie dostępna z każdego miejsca w programie, nie tylko wewnątrz funkcji. Zmienne globalne są uważane za antywzorzec, ale w celu eksperymentu możemy tak zrobić:

```
#include <stdio.h>

// x jest teraz zmienną globalną
int x;

int main()
{
        printf ("Enter X:\n");

        scanf ("%d", &x);

        printf ("You entered %d...\n", x);

        return 0;
};
```

MSVC: x86

```
_DATA    SEGMENT
COMM    _x:DWORD
$SG2456    DB    'Enter X:', 0aH, 00H
$SG2457    DB    '%d', 00H
$SG2458    DB    'You entered %d...', 0aH, 00H
_DATA    ENDS
PUBLIC    _main
EXTRN    _scanf:PROC
EXTRN    _printf:PROC
; Function compile flags: /Odtp
_TEXT    SEGMENT
_main    PROC
    push   ebp
    mov    ebp, esp
    push   OFFSET $SG2456
    call   _printf
    add    esp, 4
    push   OFFSET _x
    push   OFFSET $SG2457
    call   _scanf
    add    esp, 8
    mov    eax, DWORD PTR _x
```

```
    push   eax
    push   OFFSET $SG2458
    call   _printf
    add    esp, 8
    xor    eax, eax
    pop    ebp
    ret    0
_main    ENDP
_TEXT    ENDS
```

W tym przypadku zmienna x jest zdefiniowana w segmencie _DATA i nie następuje alokacja pamięci na stosie lokalnym. Dostęp do zmiennej jest bezpośredni, z pominięciem stosu. Niezainicjalizowane zmienne globalne nie zajmują miejsca w pliku wykonywalnym (po co alokować wyzerowaną pamięć?), dopiero w momencie odwołania pod adres zmiennej, OS alokuje blok pamięci, wypełniony zerami.

Przypiszmy teraz wprost wartość do zmiennej:

```
int x=10; // wartość domyślna
```

Otrzymamy:

```
_DATA   SEGMENT
_x      DD      0aH

...
```

Widać przypisaną wartość `0xA` typu DWORD (DD oznacza DWORD, czyli 32 bity).

Jeśli otworzysz skompilowany plik .exe w programie IDA, zobaczysz zmienną *x* na początku segmentu _DATA, a zaraz za nią ciąg znaków.

Gdybyś otworzył w programie IDA skompilowany plik .exe z poprzedniego przykładu (gdy *x* była niezainicjalizowana), zobaczyłbyś podobny wynik:

Listing 1.77: IDA

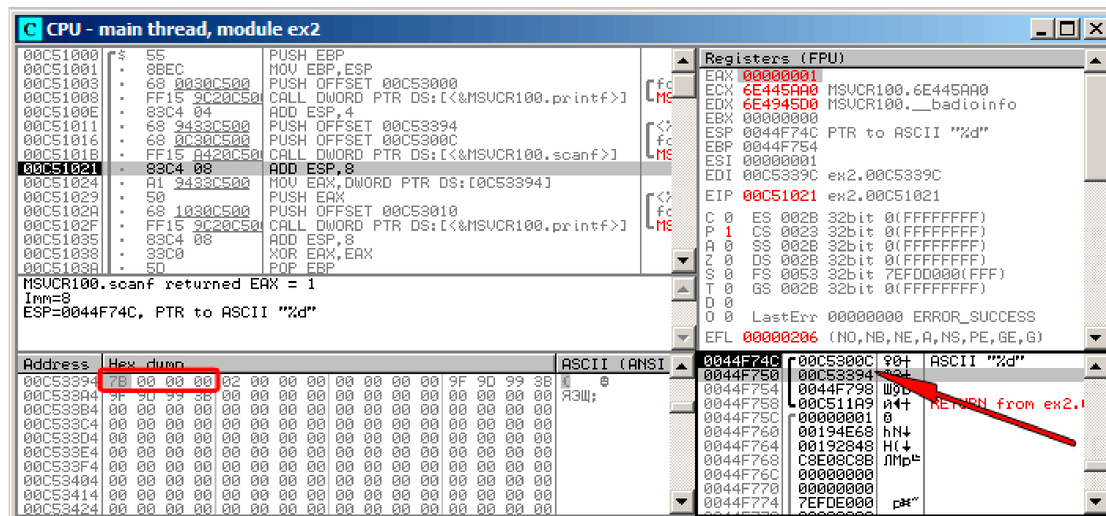
```
.data:0040FA80 _x              dd ?      ; DATA XREF: _main+10
.data:0040FA80                           ; _main+22
.data:0040FA84 dword_40FA84    dd ?      ; DATA XREF: _memset+1E
.data:0040FA84                           ; unknown_libname_1+28
.data:0040FA88 dword_40FA88    dd ?      ; DATA XREF: ___sbh_find_block+5
.data:0040FA88                           ; ___sbh_free_block+2BC
.data:0040FA8C ; LPVOID lpMem
.data:0040FA8C lpMem           dd ?      ; DATA XREF: ___sbh_find_block+B
.data:0040FA8C                           ; ___sbh_free_block+2CA
.data:0040FA90 dword_40FA90    dd ?      ; DATA XREF: _V6_HeapAlloc+13
.data:0040FA90                           ; __calloc_impl+72
.data:0040FA94 dword_40FA94    dd ?      ; DATA XREF: ___sbh_free_block+2FE
```

Zmienna _x razem z innymi zmiennymi, które nie muszą być zainicjalizowane, jest oznaczona za pomocą ?. To powoduje, że po załadowaniu pliku wykonalnego do pamięci, miejsce na te zmienne jest zaalokowane i wypełnione zerami [*ISO/IEC 9899:TC3*

(C C99 standard), (2007)6.7.8p10]. W samym pliku wykonywalnym niezainicjalizowane zmienne nie zajmują żadnego miejsca. Ma to swoje zalety, np. przy dużych tablicach.

MSVC: x86 + OllyDbg

Teraz jest nawet prościej:



Rysunek 1.16: OllyDbg: po wykonaniu `scanf()`

Zmienna jest umieszczona w segmencie danych. Po wykonaniu instrukcji PUSH (odłożenie adresu x), adres pojawia się w oknie stosu. Kliknij prawym przyciskiem na ten wiersz i wybierz „Follow in dump”. Zmienna pojawi się w oknie pamięci, po lewej stronie. Po wprowadzeniu 123 w konsoli, `0x7B` pojawi się w oknie pamięci (patrz miejsca zaznaczane na zrzucie ekranu).

Ale dlaczego pierwszy bajt to 7B? Przecież logicznie rzecz biorąc, powinno być `00 00 00 7B`! Przyczyną jest tzw. kolejność bajtów (ang. *endianess*), a x86 używa kolejności od najmniej znaczącego bajtu (tzw. *cienkokońcowość*, ang. *little-endian*). Więcej o tym przeczytasz tutaj:: **??** on page ??. Wracając do przykładu, wartość 32-bitowa jest ładowana z adresu w pamięci do EAX a następnie przekazywana do funkcji `printf()`.

x znajduje się pod adresem `0x00C53394`.

W OllyDbg można sprawdzić mapę pamięci procesu (Alt-M). Widzimy, że ten adres znajduje się w segmencie PE `.data` programu.:

Memory map

Address	Size	Owner	Section	Contains	Type	Access	Initial	Mapped as
00070000	00067000				Map	R	R	C:\Windows\System32\locale.nl
00190000	00005000			Heap	Priv	RW	RW	
00209000	00007000				Priv	RW Gua:	RW Gua:	
0044C000	00001000				Priv	RW Gua:	RW Gua:	
0044D000	00003000			Stack of main thread	Priv	RW	RW	
00590000	00007000				Priv	RW	RW	
00750000	0000C000			Default heap	Priv	RW	RW	
00C50000	00001000	ex2		PE header	Img	R	RWE Cop	
00C51000	00001000	ex2	.text	Code	Img	R E	RWE Cop	
00C52000	00001000	ex2	.rdata	Imports	Img	R	RWE Cop	
00C53000	00001000	ex2	.data	Data	Img	RW	RWE Cop	
00C54000	00001000	ex2	.reloc	Relocations	Img	R	RWE Cop	
6E3E0000	00001000	MSVCR100		PE header	Img	R	RWE Cop	
6E3E1000	000B2000	MSVCR100	.text	Code, imports, exports	Img	R E	RWE Cop	
6E493000	00006000	MSVCR100	.data	Data	Img	RW Cop	RWE Cop	
6E499000	00001000	MSVCR100	.rsrc	Resources	Img	R	RWE Cop	
6E49A000	00005000	MSVCR100	.reloc	Relocations	Img	R	RWE Cop	
755D0000	00001000	Mod_755D		PE header	Img	R	RWE Cop	
755D1000	00003000				Img	R E	RWE Cop	
755D4000	00001000				Img	RW	RWE Cop	
755D5000	00003000				Img	R	RWE Cop	
755E0000	00001000	Mod_755E		PE header	Img	R	RWE Cop	
755E1000	0004D000				Img	R E	RWE Cop	
7562E000	00005000				Img	RW Cop	RWE Cop	
75633000	00009000				Img	R	RWE Cop	
75640000	00001000	Mod_7564		PE header	Img	R	RWE Cop	
75641000	00038000				Img	R E	RWE Cop	
75679000	00002000				Img	RW	RWE Cop	
7567B000	00004000				Img	R	RWE Cop	
76F50000	00010000	kernel32		PE header	Img	R	RWE Cop	
76F60000	000D0000	kernel32	.text	Code, imports, exports	Img	R E	RWE Cop	
77030000	00010000	kernel32	.data	Data	Img	RW Cop	RWE Cop	
77040000	00010000	kernel32	.rsrc	Resources	Img	R	RWE Cop	
77050000	0000B000	kernel32	.reloc	Relocations	Img	R	RWE Cop	
77810000	00001000	KERNELBASE		PE header	Img	R	RWE Cop	
77811000	00040000	KERNELBASE	.text	Code, imports, exports	Img	R E	RWE Cop	
77851000	00002000	KERNELBASE	.data	Data	Img	RW	RWE Cop	
77853000	00001000	KERNELBASE	.rsrc	Resources	Img	R	RWE Cop	
77854000	00003000	KERNELBASE	.reloc	Relocations	Img	R	RWE Cop	
77B20000	00001000	Mod_77B2		PE header	Img	R	RWE Cop	
77B21000	00102000				Img	R E	RWE Cop	
77C23000	0002F000				Img	R	RWE Cop	
77C52000	0000C000				Img	RW Cop	RWE Cop	
77C5E000	0006B000				Img	R	RWE Cop	
77D00000	00001000	ntdll		PE header	Img	R	RWE Cop	
77D10000	000D6000	ntdll	.text	Code, exports	Img	R E	RWE Cop	
77DF0000	00001000	ntdll	RT	Code	Img	R E	RWE Cop	
77E00000	00009000	ntdll	.data	Data	Img	RW Cop	RWE Cop	

Rysunek 1.17: OllyDbg: mapa pamięci procesu

GCC: x86

Wynik kompilacji na Linuksie jest prawi taki sam, różnicą jest to, że niezainicjalizowana zmienna jest umieszczona w segmencie `_bss`. W plikach ELF[73] ten segment ma następujące atrybuty:

```
; Segment type: Uninitialized
; Segment permissions: Read/Write
```

Jeśli jednak nadasz zmiennej jakość wartość, np. 10, zostanie umieszczona w segmencie `_data`, który z kolei ma atrybuty:

```
; Segment type: Pure data
; Segment permissions: Read/Write
```

[73] Executable and Linkable Format: Format plików wykonywalnych używany w systemach z rodziny *NIX, w szczególności na Linuksie

MSVC: x64

Listing 1.78: MSVC 2012 x64

```
_DATA   SEGMENT
COMM    x:DWORD
$SG2924 DB      'Enter X:', 0aH, 00H
$SG2925 DB      '%d', 00H
$SG2926 DB      'You entered %d...', 0aH, 00H
_DATA   ENDS

_TEXT   SEGMENT
main    PROC
$LN3:
        sub     rsp, 40

        lea     rcx, OFFSET FLAT:$SG2924 ; 'Enter X:'
        call    printf
        lea     rdx, OFFSET FLAT:x
        lea     rcx, OFFSET FLAT:$SG2925 ; '%d'
        call    scanf
        mov     edx, DWORD PTR x
        lea     rcx, OFFSET FLAT:$SG2926 ; 'You entered %d...'
        call    printf

        ; zwróć 0
        xor     eax, eax

        add     rsp, 40
        ret     0
main    ENDP
_TEXT   ENDS
```

Kod jest niemal taki sam jak na x86. Zauważ, że adres zmiennej x jest przekazany do funkcji `scanf()` za pomocą instrukcji LEA, ale wartość zmiennej jest przekazywana do drugiego wywołania `printf()` za pomocą instrukcji MOV. DWORD PTR—to część kodu w asemblerze (bez związku z kodem maszynowym), pokazująca, że zmienna jest 32-bitowa i instrukcja MOV musi być odpowiednio zakodowana (opcode stosowny do rozmiaru).

ARM: Optymalizujący Keil 6/2013 (tryb Thumb)

Listing 1.79: IDA

```
.text:00000000 ; Segment type: Pure code
.text:00000000         AREA .text, CODE
...
.text:00000000 main
.text:00000000         PUSH    {R4,LR}
.text:00000002         ADR     R0, aEnterX  ; "Enter X:\n"
.text:00000004         BL      __2printf
.text:00000008         LDR     R1, =x
.text:0000000A         ADR     R0, aD       ; "%d"
```

```
.text:0000000C          BL      __0scanf
.text:00000010          LDR     R0, =x
.text:00000012          LDR     R1, [R0]
.text:00000014          ADR     R0, aYouEnteredD___ ; "You entered %d...\n"
.text:00000016          BL      __2printf
.text:0000001A          MOVS    R0, #0
.text:0000001C          POP     {R4,PC}
...
.text:00000020 aEnterX DCB "Enter X:",0xA,0 ; DATA XREF: main+2
.text:0000002A          DCB    0
.text:0000002B          DCB    0
.text:0000002C off_2C   DCD x                 ; DATA XREF: main+8
.text:0000002C                                ; main+10
.text:00000030 aD       DCB "%d",0            ; DATA XREF: main+A
.text:00000033          DCB    0
.text:00000034 aYouEnteredD___ DCB "You entered %d...",0xA,0 ; DATA XREF:
    main+14
.text:00000047          DCB 0
.text:00000047 ; .text ends
.text:00000047
...
.data:00000048 ; Segment type: Pure data
.data:00000048          AREA .data, DATA
.data:00000048          ; ORG 0x48
.data:00000048          EXPORT x
.data:00000048 x        DCD 0xA               ; DATA XREF: main+8
.data:00000048                                ; main+10
.data:00000048 ; .data ends
```

Zmienna x jest teraz globalna i z tego powodu znajduje się w innym segmencie - w segmencie danych *.data*. Można by zapytać - dlaczego w takim razie łańcuch znaków jest w segmencie kodu (*.text*)? Ponieważ x jest zmienną i z definicji jej wartość może się zmienić, co może dziać się dość często. Łańcuch znaków jest stały, nie zostanie zmieniony, więc znajduje się w segmencie *.text*.

Segment kodu czasami może znajdować się w układzie ROM[74] (pamiętaj, że mamy teraz do czynienia z systemami wbudowanymi, gdzie często występuje deficyt pamięci), a *zmienialne* zmienne w RAM[75].

Nie byłoby to racjonalne, gdybyśmy trzymali stałe zmienne w pamięci RAM, gdy dostępny jest ROM.

Co więcej, stałe zmienne w pamięci RAM powinny być zainicjalizowane przed rozpoczęciem pracy, ponieważ po włączeniu zasilania w pamięci RAM znajduję się przypadkowe dane.

Przechodząc dalej, widzimy wskaźnik z segmentu kodu (`off_2C`) do zmiennej x, i że wszystkie operacje na zmiennej zachodzą z użyciem tego wskaźnika.

Dzieje się tak, gdyż x może być w pamięci dość daleko od danej instrukcji, a więc jej adres musi być zapisany gdzieś blisko samego kodu.

[74] Pamięć tylko do odczytu (Read-Only Memory)
[75] Random-Access Memory

Instrukcja LDR w trybie Thumb może adresować zmienne w przedziale 1020 bajtów od swojej lokalizacji, a w trybie ARM —w przedziale ± 4095 bajtów.

Więc adres x musi być dość blisko, ponieważ nie ma gwarancji, że linker będzie mógł umieścić samą zmienną w pobliżu kodu. Może ona trafić nawet do zewnętrznego układu!

Jeszcze jedna rzecz: jeśli zmienna jest zadeklarowana jako *const*, kompilator Keil zaalokuje ją w segmencie `.constdata`.

Być może linker później umieści ten segment w pamięci ROM, razem z segmentem kodu.

ARM64

Listing 1.80: Nieoptymalizujący GCC 4.9.1 ARM64

```
        .comm   x,4,4
.LC0:
        .string "Enter X:"
.LC1:
        .string "%d"
.LC2:
        .string "You entered %d...\n"
f5:
; zapisz FP i LR w ramce stosu
        stp     x29, x30, [sp, -16]!
; ustaw wskaźnik ramki stosu (FP=SP)
        add     x29, sp, 0
; wczytaj wskaźnik na łańcuch znaków "Enter X:":
        adrp    x0, .LC0
        add     x0, x0, :lo12:.LC0
        bl      puts
; wczytaj wskaźnik na łańcuch znaków "%d":
        adrp    x0, .LC1
        add     x0, x0, :lo12:.LC1
; ustaw adres zmiennej globalnej x:
        adrp    x1, x
        add     x1, x1, :lo12:x
        bl      __isoc99_scanf
; ponownie ustaw adres zmiennej globalnej x:
        adrp    x0, x
        add     x0, x0, :lo12:x
; wczytaj wartość z tego adresu:
        ldr     w1, [x0]
; wczytaj wskaźnik na łańcuch znaków "You entered %d...\n"
        adrp    x0, .LC2
        add     x0, x0, :lo12:.LC2
        bl      printf
; zwróć 0
        mov     w0, 0
; przywróć FP i LR:
        ldp     x29, x30, [sp], 16
        ret
```

W tym przypadku zmienna x jest zadeklarowana jako globalna a jej adres jest wyliczany za pomocą pary instrukcji ADRP/ADD (linia 21. i 25.).

MIPS

Niezainicjalizowane zmienne globalne

x jest teraz zmienną globalną. Skompilujmy program do pliku wykonywalnego, zamiast obiektowego, i otwórzmy w programie IDA. IDA wyświetla zmienną x w sekcji .sbss pliku ELF (pamiętasz „Global Pointer”? 1.5.4 on page 34), gdyż zmienna nie jest zainicjalizowana na starcie.

Listing 1.81: Optymalizujący GCC 4.4.5 (IDA)

```
.text:004006C0 main:
.text:004006C0
.text:004006C0 var_10  = -0x10
.text:004006C0 var_4   = -4
.text:004006C0
; prolog:
.text:004006C0         lui     $gp, 0x42
.text:004006C4         addiu   $sp, -0x20
.text:004006C8         li      $gp, 0x418940
.text:004006CC         sw      $ra, 0x20+var_4($sp)
.text:004006D0         sw      $gp, 0x20+var_10($sp)
; wywołaj puts():
.text:004006D4         la      $t9, puts
.text:004006D8         lui     $a0, 0x40
.text:004006DC         jalr    $t9 ; puts
.text:004006E0         la      $a0, aEnterX     # "Enter X:" ; branch delay
    slot
; wywołaj scanf():
.text:004006E4         lw      $gp, 0x20+var_10($sp)
.text:004006E8 lui     $a0, 0x40
.text:004006EC         la      $t9, __isoc99_scanf
; przygotuj adres x:
.text:004006F0         la      $a1, x
.text:004006F4         jalr    $t9 ; __isoc99_scanf
.text:004006F8         la      $a0, aD  # "%d" ; branch delay slot
; wywołaj printf():
.text:004006FC         lw      $gp, 0x20+var_10($sp)
.text:00400700         lui     $a0, 0x40
; pobierz adres x:
.text:00400704         la      $v0, x
.text:00400708         la      $t9, printf
; załaduj wartość ze zmiennej "x" i przekaż ją do funkcji printf() przez $a1:
.text:0040070C         lw      $a1, (x - 0x41099C)($v0)
.text:00400710         jalr    $t9 ; printf
.text:00400714         la      $a0, aYouEnteredD___  # "You entered %d...\n"
    ; branch delay slot
: epilog:
.text:00400718         lw      $ra, 0x20+var_4($sp)
.text:0040071C         move    $v0, $zero
```

```
.text:00400720         jr      $ra
.text:00400724         addiu   $sp, 0x20  ; branch delay slot

...

.sbss:0041099C  # Segment type: Uninitialized
.sbss:0041099C         .sbss
.sbss:0041099C         .globl x
.sbss:0041099C x:      .space 4
.sbss:0041099C
```

IDA nie wyświetliła wszystkich informacji, więc spójrzmy na listing wygenerowany za pomocą objdump oraz komentarze:

Listing 1.82: Optymalizujący GCC 4.4.5 (objdump)

```
004006c0 <main>:
; prolog:
  4006c0:  3c1c0042   lui     gp,0x42
  4006c4:  27bdffe0   addiu   sp,sp,-32
  4006c8:  279c8940   addiu   gp,gp,-30400
  4006cc:  afbf001c   sw      ra,28(sp)
  4006d0:  afbc0010   sw      gp,16(sp)
; wywołaj puts():
  4006d4:  8f998034   lw      t9,-32716(gp)
  4006d8:  3c040040   lui     a0,0x40
  4006dc:  0320f809   jalr    t9
  4006e0:  248408f0   addiu   a0,a0,2288 ; branch delay slot
; wywołaj scanf():
  4006e4:  8fbc0010   lw      gp,16(sp)
  4006e8:  3c040040   lui     a0,0x40
  4006ec:  8f998038   lw      t9,-32712(gp)
; przygotuj adres x:
  4006f0:  8f858044   lw      a1,-32700(gp)
  4006f4:  0320f809   jalr    t9
  4006f8:  248408fc   addiu   a0,a0,2300 ; branch delay slot
; wywołaj printf():
  4006fc:  8fbc0010   lw      gp,16(sp)
  400700:  3c040040   lui     a0,0x40
; pobierz adres x:
  400704:  8f828044   lw      v0,-32700(gp)
  400708:  8f99803c   lw      t9,-32708(gp)
; załaduj wartość ze zmiennej "x" i przekaż ją do funkcji printf() przez $a1:
  40070c:  8c450000   lw      a1,0(v0)
  400710:  0320f809   jalr    t9
  400714:  24840900   addiu   a0,a0,2304 ; branch delay slot
: epilog:
  400718:  8fbf001c   lw      ra,28(sp)
  40071c:  00001021   move    v0,zero
  400720:  03e00008   jr      ra
  400724:  27bd0020   addiu   sp,sp,32   ; branch delay slot
; kilka instrukcji NOP do wyrównania początku kolejnej funkcji do granicy 16
    bajtów
  400728:  00200825   move    at,at
```

```
38  40072c:  00200825   move    at,at
```

Teraz widać, że adres zmiennej x jest wczytywany z 64KiB bufora danych za pomocą GP i ujemnego przesunięcia (linia 18). Co więcej, adresy trzech kolejnych zewnętrznych funkcji (`puts()`, `scanf()`, `printf()`) również wczytywane są z tego globalnego bufora, za pomocą GP (linia 9, 16 i 26). GP wskazuje na środek bufora, a takie przesunięcie świadczy o tym, że adresy trzech funkcji oraz zmiennej x są przechowywane gdzieś na jego początku. Ma to sens, ponieważ nasz przykład jest bardzo prosty.

Warto zauważyć, że funkcja kończy się dwiema instrukcjami NOP (`MOVE $AT,$AT` — puste instrukcje), by wyrównać początek kolejnej funkcji do granicy 16 bajtów.

Zainicjalizowane zmienne globalne

Zmieńmy nasz przykład, nadając zmiennej x wartość:

```
int x=10; // wartość domyślna
```

Teraz IDA pokazuje, że zmienna x jest wczytywana z sekcji .data:

Listing 1.83: Optymalizujący GCC 4.4.5 (IDA)

```
.text:004006A0 main:
.text:004006A0
.text:004006A0 var_10  = -0x10
.text:004006A0 var_8   = -8
.text:004006A0 var_4   = -4
.text:004006A0
.text:004006A0         lui     $gp, 0x42
.text:004006A4         addiu   $sp, -0x20
.text:004006A8         li      $gp, 0x418930
.text:004006AC         sw      $ra, 0x20+var_4($sp)
.text:004006B0         sw      $s0, 0x20+var_8($sp)
.text:004006B4         sw      $gp, 0x20+var_10($sp)
.text:004006B8         la      $t9, puts
.text:004006BC         lui     $a0, 0x40
.text:004006C0         jalr    $t9 ; puts
.text:004006C4         la      $a0, aEnterX     # "Enter X:"
.text:004006C8         lw      $gp, 0x20+var_10($sp)
; przygotuj starsze bity adresu x:
.text:004006CC         lui     $s0, 0x41
.text:004006D0         la      $t9, __isoc99_scanf
.text:004006D4         lui     $a0, 0x40
; dodaj młodsze bity adresu x:
.text:004006D8         addiu   $a1, $s0, (x - 0x410000)
; adres x jest teraz w $a1.
.text:004006DC         jalr    $t9 ; __isoc99_scanf
.text:004006E0         la      $a0, aD          # "%d"
.text:004006E4         lw      $gp, 0x20+var_10($sp)
; pobierz słowo z pamięci:
.text:004006E8         lw      $a1, x
; wartość x jest teraz w $a1.
```

```
.text:004006EC         la      $t9, printf
.text:004006F0         lui     $a0, 0x40
.text:004006F4         jalr    $t9 ; printf
.text:004006F8         la      $a0, aYouEnteredD___  # "You entered %d...\n"
.text:004006FC         lw      $ra, 0x20+var_4($sp)
.text:00400700         move    $v0, $zero
.text:00400704         lw      $s0, 0x20+var_8($sp)
.text:00400708         jr      $ra
.text:0040070C         addiu   $sp, 0x20

...

.data:00410920         .globl x
.data:00410920 x:      .word 0xA
```

Dlaczego nie z .sdata? Być może wpływają na to pewne opcje GCC?

W każdym razie, x jest w .data, a jest to pamięć współdzielona. Możemy zobaczyć jak taka pamięć jest wykorzystywana.

Adres zmiennej jest konstruowany za pomocą pary instrukcji.

W naszym przykładzie są to LUI („Load Upper Immediate”) i ADDIU („Add Immediate Unsigned Word”).

Poniżej listing wygenerowany za pomocą objdump, do bardziej szczegółowej analizy:

Listing 1.84: Optymalizujący GCC 4.4.5 (objdump)

```
004006a0 <main>:
  4006a0:  3c1c0042   lui     gp,0x42
  4006a4:  27bdffe0   addiu   sp,sp,-32
  4006a8:  279c8930   addiu   gp,gp,-30416
  4006ac:  afbf001c   sw      ra,28(sp)
  4006b0:  afb00018   sw      s0,24(sp)
  4006b4:  afbc0010   sw      gp,16(sp)
  4006b8:  8f998034   lw      t9,-32716(gp)
  4006bc:  3c040040   lui     a0,0x40
  4006c0:  0320f809   jalr    t9
  4006c4:  248408d0   addiu   a0,a0,2256
  4006c8:  8fbc0010   lw      gp,16(sp)
; przygotuj starsze bity adresu x:
  4006cc:  3c100041   lui     s0,0x41
  4006d0:  8f998038   lw      t9,-32712(gp)
  4006d4:  3c040040   lui     a0,0x40
; dodaj młodsze bity adresu x:
  4006d8:  26050920   addiu   a1,s0,2336
; adres x jest teraz w $a1.
  4006dc:  0320f809   jalr    t9
  4006e0:  248408dc   addiu   a0,a0,2268
  4006e4:  8fbc0010   lw      gp,16(sp)
; starsza część adresu wciąż jest w $s0.
; dodaj młodsze bity i pobierz słowo z pamięci:
  4006e8:  8e050920   lw      a1,2336(s0)
; wartość x jest teraz w $a1.
```

```
  4006ec:  8f99803c   lw      t9,-32708(gp)
  4006f0:  3c040040   lui     a0,0x40
  4006f4:  0320f809   jalr    t9
  4006f8:  248408e0   addiu   a0,a0,2272
  4006fc:  8fbf001c   lw      ra,28(sp)
  400700:  00001021   move    v0,zero
  400704:  8fb00018   lw      s0,24(sp)
  400708:  03e00008   jr      ra
  40070c:  27bd0020   addiu   sp,sp,32
```

Widać, że po zbudowaniu adresu za pomocą LUI i ADDIU starsze bity wciąż znajdują się w rejestrze $S0. Można teraz zakodować przesunięcie w instrukcji LW („Load Word”), by za pomocą tylko tej jednej instukcji załadować zmienną z pamięci i przekazać do funkcji printf().

Wiemy już, że rejestry przechowujące tymczasowe dane mają prefiks T-. W przykładzie widzimy również takie, które rozpoczynają się od S- — są to rejestry, których zawartość musi zostać zachowana, jeśli funkcja planuje ich użyć (np. muszą być gdzieś zapisana, a później przywrócone). Dzięki temu, gdy wywołujemy daną funkcję, to po jej zakończeniu i powrocie sterowania dane w tych rejestrach będą takie same, jak przed jej wywołaniem.

Dlatego wartość $S0 została ustawiona w adresie 0x4006cc, a następnie ponownie użyta pod adresem 0x4006e8, po wywołaniu funkcji scanf(). Funkcja scanf() nie zmieniła tej wartości.

1.12.4 scanf()

Jak wspomniano wcześniej, używanie scanf() w dzisiejszych czasach jest nieco staroświeckie. Jeśli jednak musisz to zrobić, należy się upewnić czy wykonanie scanf() zakończyło się poprawnie, bez żadnego błędu.

```
#include <stdio.h>

int main()
{
        int x;
        printf ("Enter X:\n");

        if (scanf ("%d", &x)==1)
                printf ("You entered %d...\n", x);
        else
                printf ("What you entered? Huh?\n");

        return 0;
};
```

Zgodnie ze standardem scanf()[76] zwraca liczbę pól, które zostały z sukcesem wczytane i zapisane.

[76] scanf, wscanf: MSDN

W naszym przypadku, jeśli wszystko pójdzie dobrze i użytkownik wprowadził liczbę, `scanf()` zwróci 1. W przypadku wystąpienia błędu (lub EOF[77]), zwróci 0.

Dodaliśmy więcej kodu by sprawdzić co zwraca `scanf()` i wypiszmy ewentualne komunikaty o błędach.

Poniżej pokazano program w działaniu:

```
C:\...>ex3.exe
Enter X:
123
You entered 123...

C:\...>ex3.exe
Enter X:
ouch
What you entered? Huh?
```

MSVC: x86

Poniżej wynik kompilacji pod MSVC 2010:

```
        lea     eax, DWORD PTR _x$[ebp]
        push    eax
        push    OFFSET $SG3833 ; '%d', 00H
        call    _scanf
        add     esp, 8
        cmp     eax, 1
        jne     SHORT $LN2@main
        mov     ecx, DWORD PTR _x$[ebp]
        push    ecx
        push    OFFSET $SG3834 ; 'You entered %d...', 0aH, 00H
        call    _printf
        add     esp, 8
        jmp     SHORT $LN1@main
$LN2@main:
        push    OFFSET $SG3836 ; 'What you entered? Huh?', 0aH, 00H
        call    _printf
        add     esp, 4
$LN1@main:
        xor     eax, eax
```

Funkcja wywołująca (`main()`) potrzebuje rezultatu zwróconego przez funkcję wywoływaną (`scanf()`), więc funkcja wywoływana zwraca go za pomocą rejestru EAX.

Rezultat sprawdzamy za pomocą instrukcji `CMP EAX, 1` (*CoMPare*) — porównujemy wartość w rejestrze EAX z liczbą 1.

Instrukcja `JNE` to skok warunkowy, następujący po `CMP`. `JNE` oznacza *Jump if Not Equal*.

[77] End of File

Jeśli wartość w EAX jest różna od 1, CPU przekaże sterowanie pod adres z operandu instrukcji JNE, w naszym przypadku jest to $LN2@main. Przekazanie sterowania pod ten adres oznacza wykonanie funkcji printf() z argumentem What you entered? Huh?. Jeśli natomiast scanf() zakończyła się sukcesem i wartość w EAX jest równa 1, skok warunkowy nie zostanie wykonany i kolejno zostanie wywołana funkcja printf(), z dwoma argumentami:
'You entered %d...' i wartością x.

W tym drugin przypadku - gdy scanf() zakończyła się poprawnie - nie ma potrzeby wykonywać drugiego wywołania funkcji printf(), stąd przed wywołaniem znajduje się instrukcja JMP (skok bezwarunkowy). Instrukcja przekazuje sterowanie w miejsce za drugim wywołaniem printf(), ale przed instrukcją XOR EAX, EAX, która realizuje return 0.

Można powiedzieć, że porówywanie dwóch wartości jest zwykle realizowane przez parę instrukcji CMP/Jcc, gdzie *cc* oznacza *condition code* (*kod warunku*). CMP porównuje dwie wartości i ustawia flagę procesora [78]. Jcc sprawdza te flagi i decyduje czy przekazać sterowanie pod podany adres.

Zabrzmi to paradoksalnie, ale instrukcja CMP to tak na prawdę SUB (subtract - *odejmij*). Nie tylko CMP, ale wszystkie instrukcje arytmetyczne modyfikują flagi procesora. Jeśli porównamy 1 z 1, $1 - 1$ daje 0, więc flaga ZF zostanie ustawiona. W żadnym innym przypadku flaga ZF nie zostanie ustawiona, poza tym gdy operandy są sobie równe. JNE sprawdza tylko flagę ZF i wykonuje skok, jeśli nie jest ustawiona. JNE jest synonimem JNZ (*Jump if Not Zero*). Asembler tłumaczy zarówno JNE jak i JNZ na ten sam kod operacji (opcode). Instrukcja CMP może być zastąpiona przez SUB i prawie wszystko powinno działać poprawnie, poza tym, że SUB zmieni wartość pierwszego operandu na wynik operacji odejmowania. CMP to *SUB bez zapisywania wyniku operacji, ale ze zmianą flag*.

MSVC: x86: IDA

Nadszedł czas na uruchomienie programu IDA i pokazanie jego możliwości. Przy okazji, początkującym pomoże ustawienie opcji /MD w MSVC, co spowoduje, że wszystkie funkcji biblioteki standardowej nie będą statycznie zlinkowane do pliku wykonywalnego, ale zostaną zaimportowane z MSVCR*.DLL podczas wykonania. Dzięki temu łatwiej będzie zobaczyć, które funkcje z biblioteki standardowej zostały użyte i gdzie.

Podczas analizy kodu w programie IDA warto dla siebie (i innych) robić notatki. W tym przypadku widzimy, że skok JNZ wykona się w przypadku błędu. Można przesunąć kursor do etykiety, nacisnąć „n” i zmienić nazwę na „error”. Zmienimy również nazwę kolejnej etykiety na „exit”.

Poniżej listing po zmianiach nazw:

```
.text:00401000 _main proc near
.text:00401000
.text:00401000 var_4 = dword ptr -4
.text:00401000 argc  = dword ptr  8
.text:00401000 argv  = dword ptr  0Ch
.text:00401000 envp  = dword ptr  10h
```

[78] rejestr FLAGS, więcej o tym przeczytasz pod adresem: wikipedia.

```
.text:00401000
.text:00401000       push    ebp
.text:00401001       mov     ebp, esp
.text:00401003       push    ecx
.text:00401004       push    offset Format ; "Enter X:\n"
.text:00401009       call    ds:printf
.text:0040100F       add     esp, 4
.text:00401012       lea     eax, [ebp+var_4]
.text:00401015       push    eax
.text:00401016       push    offset aD ; "%d"
.text:0040101B       call    ds:scanf
.text:00401021       add     esp, 8
.text:00401024       cmp     eax, 1
.text:00401027       jnz     short error
.text:00401029       mov     ecx, [ebp+var_4]
.text:0040102C       push    ecx
.text:0040102D       push    offset aYou ; "You entered %d...\n"
.text:00401032       call    ds:printf
.text:00401038       add     esp, 8
.text:0040103B       jmp     short exit
.text:0040103D
.text:0040103D error: ; CODE XREF: _main+27
.text:0040103D       push    offset aWhat ; "What you entered? Huh?\n"
.text:00401042       call    ds:printf
.text:00401048       add     esp, 4
.text:0040104B
.text:0040104B exit: ; CODE XREF: _main+3B
.text:0040104B       xor     eax, eax
.text:0040104D       mov     esp, ebp
.text:0040104F       pop     ebp
.text:00401050       retn
.text:00401050 _main endp
```

Te drobne modyfikacje ułatwiły zrozumienie kodu, jednak nie warto przesadzać i komentować każdej instrukcji.

W IDA nożesz również ukryć (zwinąć) kod wybranej funkcji. Zaznacz blok kodu, wciśnij Ctrl-„–” na klawiaturze numerycznej i wpisz tekst, który ma zostać wyświetlony zamiast kodu.

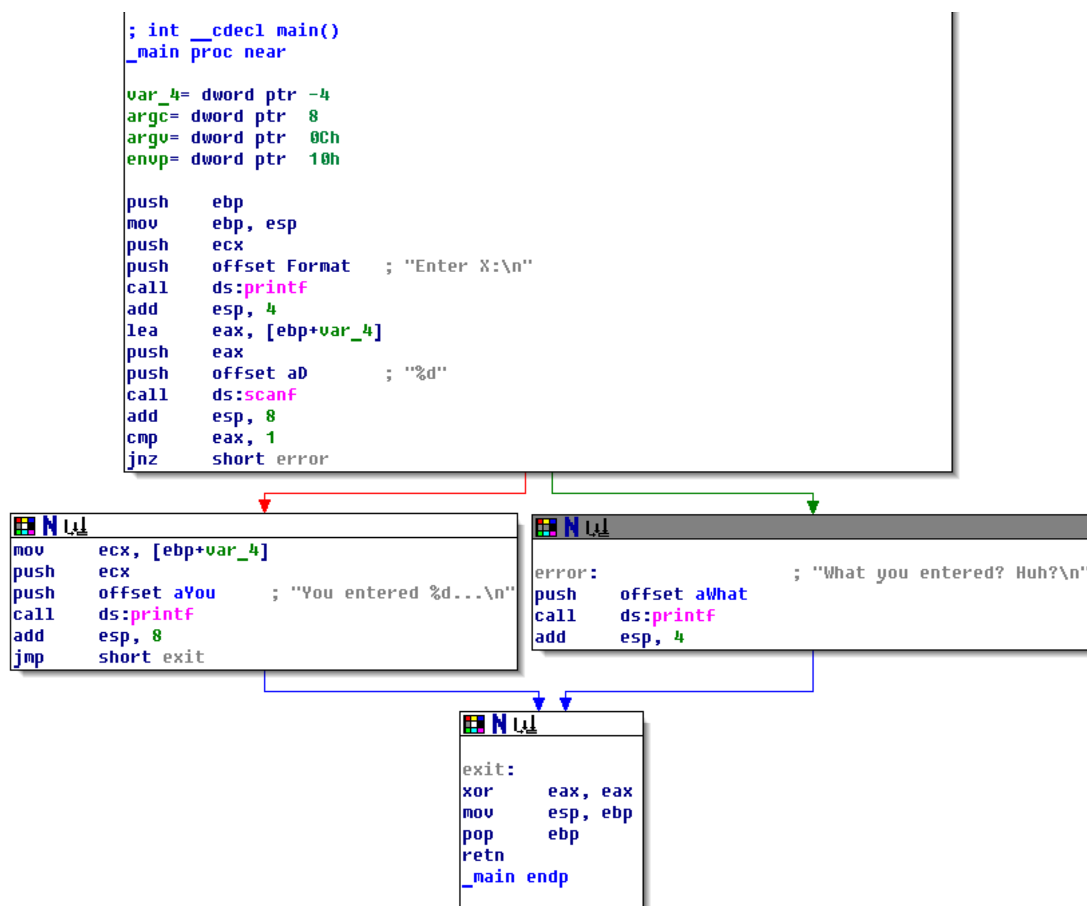
Ukryjmy dwa bloki kodu i nadajmy im nazwy:

```
.text:00401000 _text segment para public 'CODE' use32
.text:00401000       assume cs:_text
.text:00401000       ;org 401000h
.text:00401000 ; ask for X
.text:00401012 ; get X
.text:00401024       cmp  eax, 1
.text:00401027       jnz  short error
.text:00401029 ; print result
.text:0040103B       jmp  short exit
.text:0040103D
.text:0040103D error: ; CODE XREF: _main+27
.text:0040103D       push offset aWhat ; "What you entered? Huh?\n"
```

```
.text:00401042       call ds:printf
.text:00401048       add  esp, 4
.text:0040104B
.text:0040104B exit: ; CODE XREF: _main+3B
.text:0040104B       xor  eax, eax
.text:0040104D       mov  esp, ebp
.text:0040104F       pop  ebp
.text:00401050       retn
.text:00401050 _main endp
```

By rozwinąć poprzednio zwinięte fragmenty, użyj Ctrl-„+” na klawiaturze numerycznej.

Po naciśnięciu „spacji” zobaczymy reprezentację funkcji w postaci grafu.



Rysunek 1.18: Tryb grafu w IDA

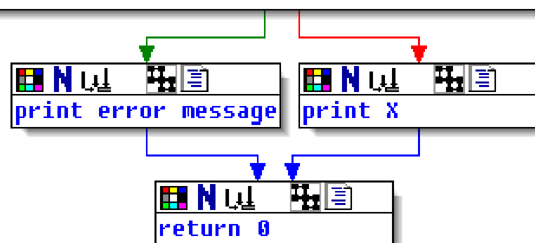
Z każdego skoku warunkowego wychodzą dwie strzałki: zielona i czerwona. Zielona wskazuje blok, który się wykona w przypadku wykonania skoku, a czerwona - blok, który się wykona, gdy do skoku nie dojdzie.

W tym trybie można zwinąć węzły i nadać nazwę tak stworzonej „grupie węzłów”. Zróbmy to dla 3 bloków:

```
; int __cdecl main()
_main proc near

var_4= dword ptr -4
argc= dword ptr  8
argv= dword ptr  0Ch
envp= dword ptr  10h

push    ebp
mov     ebp, esp
push    ecx
push    offset Format   ; "Enter X:\n"
call    ds:printf
add     esp, 4
lea     eax, [ebp+var_4]
push    eax
push    offset aD       ; "%d"
call    ds:scanf
add     esp, 8
cmp     eax, 1
jnz     short error
```

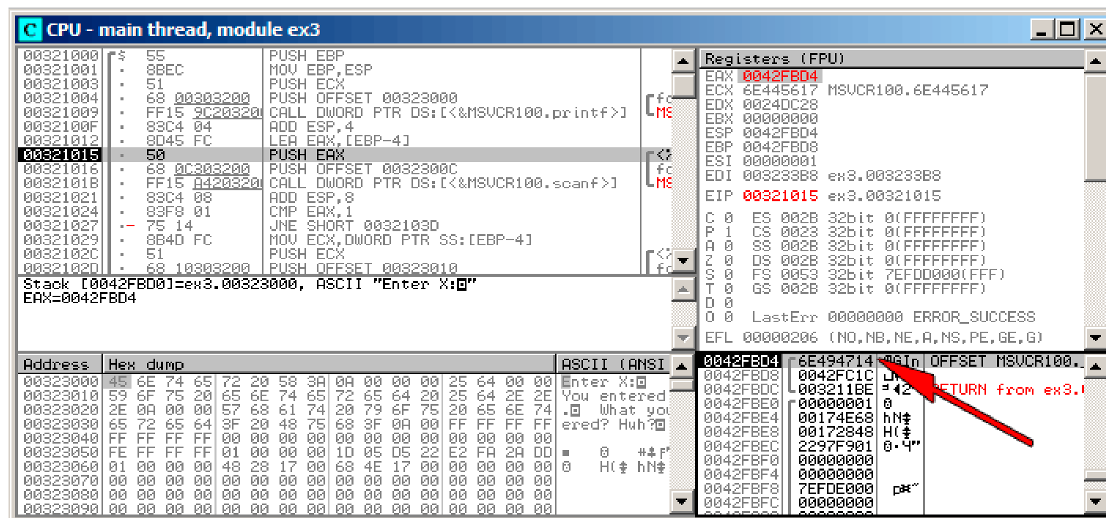


Rysunek 1.19: Tryb grafu w IDA przy 3 zwiniętych węzłach

Jest to dość użyteczne. Można powiedzieć, że istotną częścią pracy osoby zajmującej się inżynierią wsteczną (a także każdego innego badacza) jest ograniczenie ilości informacji.

MSVC: x86 + OllyDbg

Spróbujmy zhackować nasz program w OllyDbg, zmuszając go, by uznał, że funkcja `scanf()` wykonała się bez błędów. Kiedy adres zmiennej lokalnej jest przekazywany do `scanf()`, zmienna początkowo zawiera przypadkową wartość, w tym wypadku `0x6E494714`:



Rysunek 1.20: OllyDbg: przekazywanie adresu zmiennej do `scanf()`

Kiedy wykonywana jest funkcja `scanf()`, w konsoli wpiszmy coś, co z pewnością nie jest liczbą, na przykład „asdasd”. `scanf()` kończy działanie z 0 w EAX, co wskazuje na wystąpienie błędu.

Możemy sprawdzić wartość zmiennej lokalnej na stosie i zauważyć, że się ona nie zmieniła. W rzeczy samej, dlaczego funkcja `scanf()` miałaby cokolwiek tam zapisać? Jej wykonanie nie spowodowało nic, poza zwróceniem zera.

Spróbujmy „zhackować” nasz program. Kliknij prawym przyciskiem na EAX, wśród opcji znajduje się „Set to 1” (*ustaw na 1*). To jest to, czego szukamy.

Mamy teraz 1 w EAX, a więc kolejne sprawdzenie powinno się wykonać zgodnie z oczekiwaniami i `printf()` powinna wyświetlić wartość zmiennej ze stosu.

Po wznowieniu wykonania programu (F9) widzimy następujący efekt w oknie konsoli:

Listing 1.85: console window

```
Enter X:
asdasd
You entered 1850296084...
```

1850296084 to postać dziesiętna liczby, którą widzieliśmy na stosie (`0x6E494714`)!

MSVC: x86 + Hiew

Na tym przykładzie pokażemy proste *poprawianie* plików wykonalnych. Tak zmodyfikujmy program, by zawsze wypisał wejście wprowadzony przez użytkownika, niezależnie od jego treści.

Zakładając, że plik wykonywalny jest linkowany dynamicznie z `MSVCR*.DLL` (kompilacja z opcją `/MD`), zobaczymy funkcję `main()` na początku sekcji `.text`. Otwórzmy plik w Hiew i znajdźmy początek sekcji `.text` (Enter, F8, F6, Enter, Enter).

Widzimy:

```
Hiew: ex3.exe
    C:\Polygon\ollydbg\ex3.exe                    FRO --------          a32 PE .00401000 Hie
.00401000: 55                                 push     ebp
.00401001: 8BEC                               mov      ebp,esp
.00401003: 51                                 push     ecx
.00401004: 6800304000                         push     000403000 ;'Enter X:' --1
.00401009: FF1594204000                       call     printf
.0040100F: 83C404                             add      esp,4
.00401012: 8D45FC                             lea      eax,[ebp][-4]
.00401015: 50                                 push     eax
.00401016: 680C304000                         push     00040300C --2
.0040101B: FF158C204000                       call     scanf
.00401021: 83C408                             add      esp,8
.00401024: 83F801                             cmp      eax,1
.00401027: 7514                               jnz     .00040103D --3
.00401029: 8B4DFC                             mov      ecx,[ebp][-4]
.0040102C: 51                                 push     ecx
.0040102D: 6810304000                         push     000403010 ;'You entered %d...'
.00401032: FF1594204000                       call     printf
.00401038: 83C408                             add      esp,8
.0040103B: EB0E                               jmps    .00040104B --5
.0040103D: 6824304000                        3push     000403024 ;'What you entered?
.00401042: FF1594204000                       call     printf
.00401048: 83C404                             add      esp,4
.0040104B: 33C0                              5xor      eax,eax
.0040104D: 8BE5                               mov      esp,ebp
.0040104F: 5D                                 pop      ebp
.00401050: C3                                 retn ; -^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-
.00401051: B84D5A0000                         mov      eax,000005A4D ;'  ZM'
1Global 2FilBlk 3CryBlk 4ReLoad 5OrdLdr 6String 7Direct 8Table 9 1byte 10Leave 11Nak
```

Rysunek 1.21: Hiew: funkcja `main()`

Hiew znajduje łańuchy znaków ASCIIZ[79] i je wyświetla, tak samo dzieje się również z nazwami zaimportowanych funkcji.

[79] ASCII Zero ( )

Przesuń kursor do adresu `.00401027` (znajduje się tam instrukcja JNZ, którą musimy ominąć), naciśnij F3 i wpisz „9090” (co oznacza dwie instrukcje NOP):

```
Hiew: ex3.exe
   C:\Polygon\ollydbg\ex3.exe                  FWO EDITMODE            a32 PE  0000
 00000400: 55                                  push       ebp
 00000401: 8BEC                                mov        ebp,esp
 00000403: 51                                  push       ecx
 00000404: 6800304000                          push       000403000 ;' @0 '
 00000409: FF1594204000                        call       d,[000402094]
 0000040F: 83C404                              add        esp,4
 00000412: 8D45FC                              lea        eax,[ebp][-4]
 00000415: 50                                  push       eax
 00000416: 680C304000                          push       00040300C ;' @0'
 0000041B: FF158C204000                        call       d,[00040208C]
 00000421: 83C408                              add        esp,8
 00000424: 83F801                              cmp        eax,1
 00000427: 90                                  nop
 00000428: 90                                  nop
 00000429: 8B4DFC                              mov        ecx,[ebp][-4]
 0000042C: 51                                  push       ecx
 0000042D: 6810304000                          push       000403010 ;' @0'
 00000432: FF1594204000                        call       d,[000402094]
 00000438: 83C408                              add        esp,8
 0000043B: EB0E                                jmps       00000044B
 0000043D: 6824304000                          push       000403024 ;' @0$'
 00000442: FF1594204000                        call       d,[000402094]
 00000448: 83C404                              add        esp,4
 0000044B: 33C0                                xor        eax,eax
 0000044D: 8BE5                                mov        esp,ebp
 0000044F: 5D                                  pop        ebp
 00000450: C3                                  retn ; -^-^-^-^-^-^-^-^-^-^-^-^-^-^-
 1      2NOPs   3      4      5      6      7      8Table  9      10
```



Rysunek 1.22: Hiew: zastąpienie JNZ przez dwie instrukcje NOP

Następnie naciśnij F9 (update). Plik wykonywalny został zapisany na dysk i będzie się zachowywał zgodnie z naszymi oczekiwaniami.

Dwie instrukcje NOP nie są najbardziej eleganckim rozwiązaniem. Innym sposobem byłoby poprawienie instrukcji przez zapisanie 0 do drugiego bajtu kodu operacji (przesunięcie skoku), by JNZ zawsze skakała do kolejnej instrukcji.

Można też program zmodyfikować w drugą stronę: zastąpić pierwszy bajt przez EB, nie zmieniając drugiego bajtu (przesunięcie skoku). Otrzymamy wtedy skok bezwarunkowy, który zawsze będzie zachodził, przez co za każdym razem dostaniemy wiadomość o błędzie.

MSVC: x64

Pracujemy ze zmiennymi typu *int*, które na x86-64 wciaż będą 32-bitowe, stąd w kodzie zobaczymy wykorzystanie 32-bitowych części rejestrów (z prefiksem E-). Jednak przy pracy ze wskaźnikami będą używane 64-bitowe rejestry, z prefiksem R-.

Listing 1.86: MSVC 2012 x64

```
_DATA   SEGMENT
$SG2924 DB      'Enter X:', 0aH, 00H
$SG2926 DB      '%d', 00H
$SG2927 DB      'You entered %d...', 0aH, 00H
$SG2929 DB      'What you entered? Huh?', 0aH, 00H
_DATA   ENDS

_TEXT   SEGMENT
x$ = 32
main    PROC
$LN5:
        sub     rsp, 56
        lea     rcx, OFFSET FLAT:$SG2924 ; 'Enter X:'
        call    printf
        lea     rdx, QWORD PTR x$[rsp]
        lea     rcx, OFFSET FLAT:$SG2926 ; '%d'
        call    scanf
        cmp     eax, 1
        jne     SHORT $LN2@main
        mov     edx, DWORD PTR x$[rsp]
        lea     rcx, OFFSET FLAT:$SG2927 ; 'You entered %d...'
        call    printf
        jmp     SHORT $LN1@main
$LN2@main:
        lea     rcx, OFFSET FLAT:$SG2929 ; 'What you entered? Huh?'
        call    printf
$LN1@main:
        ; zwróć 0
        xor     eax, eax
        add     rsp, 56
        ret     0
main    ENDP
_TEXT   ENDS
END
```

ARM

ARM: Optymalizujący Keil 6/2013 (tryb Thumb)

Listing 1.87: Optymalizujący Keil 6/2013 (tryb Thumb)

```
var_8    = -8

         PUSH    {R3,LR}
         ADR     R0, aEnterX     ; "Enter X:\n"
```

```
          BL      __2printf
          MOV     R1, SP
          ADR     R0, aD          ; "%d"
          BL      __0scanf
          CMP     R0, #1
          BEQ     loc_1E
          ADR     R0, aWhatYouEntered ; "What you entered? Huh?\n"
          BL      __2printf

loc_1A                            ; CODE XREF: main+26
          MOVS    R0, #0
          POP     {R3,PC}

loc_1E                            ; CODE XREF: main+12
          LDR     R1, [SP,#8+var_8]
          ADR     R0, aYouEnteredD___ ; "You entered %d...\n"
          BL      __2printf
          B       loc_1A
```

Nowymi instrukcjami są CMP i BEQ[80].

CMP jest równoważna instrukcji o takiej samej nazwie z x86. Odejmuje ona jeden argument od drugiego (bez zapisywania wyniku) i ustawia odpowiednie flagi procesora.

BEQ skacze pod inny adres, jeśli wartość flagi Z jest równa 1. Taka sytuacja w naszym przykładzie zajdzie, jeśli wartość rejestru i liczby z operandów instrukcji CMP będą sobie równe (tym samym ich różnica będzie równa 0). Tak samo zachowuje się instrukcja JZ na x86.

Cała reszta kodu jest bardzo prosta: przepływ sterowania dzieli się na dwa rozgałęzienia, które spotykają się w punkcie, gdzie 0 jest zapisywane do rejestru R0 jako wartość zwracana. Następnie funkcja się kończy.

ARM64

Listing 1.88: Nieoptymalizujący GCC 4.9.1 ARM64

```
.LC0:
        .string "Enter X:"
.LC1:
        .string "%d"
.LC2:
        .string "You entered %d...\n"
.LC3:
        .string "What you entered? Huh?"
f6:
; zapisz FP i LR w ramce stosu:
        stp     x29, x30, [sp, -32]!
; ustaw wskaźnik ramki stosu (FP=SP)
        add     x29, sp, 0
```

[80](PowerPC, ARM) Branch if Equal

```
; załaduj wskaźnik na łańcuch znaków "Enter X:":
        adrp    x0, .LC0
        add     x0, x0, :lo12:.LC0
        bl      puts
; załaduj wskaźnik na łańcuch znaków "%d":
        adrp    x0, .LC1
        add     x0, x0, :lo12:.LC1
; oblicz adres zmiennej x na stosie lokalnym:
        add     x1, x29, 28
        bl      __isoc99_scanf
; scanf() zwraca wynik przez rejestr W0
; sprawdź go:
        cmp     w0, 1
; BNE oznacza Branch if Not Equal
; jeśli W0<>1, skocz do L2
        bne     .L2
; w tym miejscu W0=1, co oznacza brak błędu
; załaduj wartość x ze stosu lokalnego
        ldr     w1, [x29,28]
; załaduj wskaźnik na łańcuch znaków "You entered %d...\n":
        adrp    x0, .LC2
        add     x0, x0, :lo12:.LC2
        bl      printf
; przeskocz przez kod wyświetlający "What you entered? Huh?":
        b       .L3
.L2:
; załaduj wskaźnik na łańcuch znaków "What you entered? Huh?":
        adrp    x0, .LC3
        add     x0, x0, :lo12:.LC3
        bl      puts
.L3:
; zwróć 0
        mov     w0, 0
; przywróć FP i LR:
        ldp     x29, x30, [sp], 32
        ret
```

Przepływ sterowania w tym przypadku rozgałęzia się dzięki parze instrukcji CMP/BNE (Branch if Not Equal).

MIPS

Listing 1.89: Optymalizujący GCC 4.4.5 (IDA)

```
.text:004006A0 main:
.text:004006A0
.text:004006A0 var_18    = -0x18
.text:004006A0 var_10    = -0x10
.text:004006A0 var_4     = -4
.text:004006A0
.text:004006A0           lui     $gp, 0x42
.text:004006A4           addiu   $sp, -0x28
.text:004006A8           li      $gp, 0x418960
```

```
.text:004006AC          sw      $ra, 0x28+var_4($sp)
.text:004006B0          sw      $gp, 0x28+var_18($sp)
.text:004006B4          la      $t9, puts
.text:004006B8          lui     $a0, 0x40
.text:004006BC          jalr    $t9 ; puts
.text:004006C0          la      $a0, aEnterX     # "Enter X:"
.text:004006C4          lw      $gp, 0x28+var_18($sp)
.text:004006C8          lui     $a0, 0x40
.text:004006CC          la      $t9, __isoc99_scanf
.text:004006D0          la      $a0, aD          # "%d"
.text:004006D4          jalr    $t9 ; __isoc99_scanf
.text:004006D8          addiu   $a1, $sp, 0x28+var_10  # branch delay slot
.text:004006DC          li      $v1, 1
.text:004006E0          lw      $gp, 0x28+var_18($sp)
.text:004006E4          beq     $v0, $v1, loc_40070C
.text:004006E8          or      $at, $zero       # branch delay slot, NOP
.text:004006EC          la      $t9, puts
.text:004006F0          lui     $a0, 0x40
.text:004006F4          jalr    $t9 ; puts
.text:004006F8          la      $a0, aWhatYouEntered  # "What you entered?
    Huh?"
.text:004006FC          lw      $ra, 0x28+var_4($sp)
.text:00400700          move    $v0, $zero
.text:00400704          jr      $ra
.text:00400708          addiu   $sp, 0x28

.text:0040070C loc_40070C:
.text:0040070C          la      $t9, printf
.text:00400710          lw      $a1, 0x28+var_10($sp)
.text:00400714          lui     $a0, 0x40
.text:00400718          jalr    $t9 ; printf
.text:0040071C          la      $a0, aYouEnteredD___  # "You entered
    %d...\n"
.text:00400720          lw      $ra, 0x28+var_4($sp)
.text:00400724          move    $v0, $zero
.text:00400728          jr      $ra
.text:0040072C          addiu   $sp, 0x28
```

`scanf()` zwraca wynik wykonania przez rejestr $V0. Jest on sprawdzany pod adresem 0x004006E4, przez porównanie jego wartości z $V1 (wartość 1 została zapisana do $V1 wcześniej, pod adresem 0x004006DC). BEQ oznacza „Branch Equal”. Jeśli dwie wartości są sobie równe (w naszym przykładzie tak będzie w przypadku sukcesu), sterowanie skacze pod adres 0x0040070C.

Ćwiczenie

Jak widać, instrukcja JNE/JNZ może być łatwo zastąpiona przez JE/JZ i vice versa (a BNE przez BEQ i vice versa). Jednak należy pamiętać o zamianie miejscami bloków kodu do wykonania. Spróbuj to zrobić w ramach ćwiczeń.

1.12.5 Ćwiczenie

- http://challenges.re/53

1.13 Warto zauważyć: zmienne globalne vs zmienne lokalne

Teraz już wiesz, że zmienne globalne są wypełniane zerami przez OS przy starcie programu (1.12.3 on page 103, [*ISO/IEC 9899:TC3 (C C99 standard)*, (2007)6.7.8p10]), a zmienne lokalne nie (1.9.4 on page 52).

Czasami możesz zapomnieć zainicjalizować zmienną globalną, a program polega na tym, że jej wartość na starcie wynosi 0. Następnie zmieniasz program i przenosisz zmienną do funkcji, zmieniając ją na lokalną. Jednak tym razem jej wartość nie będzie wynosiła 0, co może prowadzić do nieprzyjemnych błędów.

1.14 Dostęp do przekazanych argumentów

Poznaliśmy już jak funkcja wywołująca przekazuje argumenty przez stos do funkcji wywoływanej. Ale w jaki sposób funkcja wywoływana może się do nich dostać?

Listing 1.90: Prosty przykład

```
#include <stdio.h>

int f (int a, int b, int c)
{
        return a*b+c;
};

int main()
{
        printf ("%d\n", f(1, 2, 3));
        return 0;
};
```

1.14.1 x86

MSVC

Poniżej wynik kompilacji (MSVC 2010 Express):

Listing 1.91: MSVC 2010 Express

```
_TEXT   SEGMENT
_a$ = 8         ; size = 4
_b$ = 12        ; size = 4
_c$ = 16        ; size = 4
_f      PROC
        push    ebp
```

```
        mov     ebp, esp
        mov     eax, DWORD PTR _a$[ebp]
        imul    eax, DWORD PTR _b$[ebp]
        add     eax, DWORD PTR _c$[ebp]
        pop     ebp
        ret     0
_f      ENDP

_main   PROC
        push    ebp
        mov     ebp, esp
        push    3 ; 3rd argument
        push    2 ; 2nd argument
        push    1 ; 1st argument
        call    _f
        add     esp, 12
        push    eax
        push    OFFSET $SG2463 ; '%d', 0aH, 00H
        call    _printf
        add     esp, 8
        ; return 0
        xor     eax, eax
        pop     ebp
        ret     0
_main   ENDP
```

Na listingu widać, jak funkcja `main()` odkłada na stos 3 liczby i wywołuje `f(int,int,int)`.

Dostęp do argumentów `f()` uzyskuje za pomocą makr, jak np.:
`_a$ = 8`, podobnie jak do zmiennych lokalnych, ale z dodatnim przesunięciem. Niejako adresujemy pamięć *poza stosem*, gdyż stos rośnie w dół, a my dodajemy wartość dodatnią _a$ do rejestru EBP (wskaźnik ramki stosu).

Następnie wartość a jest zapisywana do EAX. Po wykonaniu instrukcji IMUL, wartość w EAX jest iloczynem wartości z EAX i wartości wskazywanej przez przesunięcie _b.

Kolejno wykonywana jest instrukcja ADD, która dodaje wartość pokazywaną przez przesunięcie _c do EAX.

Wartość w EAX już nie musi być nigdzie zapisywana, gdyż jest to wynik funkcji, a w tej konwencji wywoływania jest on zwracany przez rejestr EAX. Po powrocie funkcja wywołująca pobiera wartość z EAX i używa jako argumentu do `printf()`.

MSVC + OllyDbg

Prześledźmy działanie programu w OllyDbg. Gdy zatrzymamy się na pierwszej instrukcji w `f()`, używającej jednego z argumentów (pierwszego), widać, że EBP pokazuje na ramkę stosu (oznaczona czerwonym prostokątem).

Pierwszym elementem w ramce stosu jest zapisana wartość EBP, drugim jest RA (adres powrotu), trzecim jest pierwszy argument funkcji, następnie drugi i trzeci.

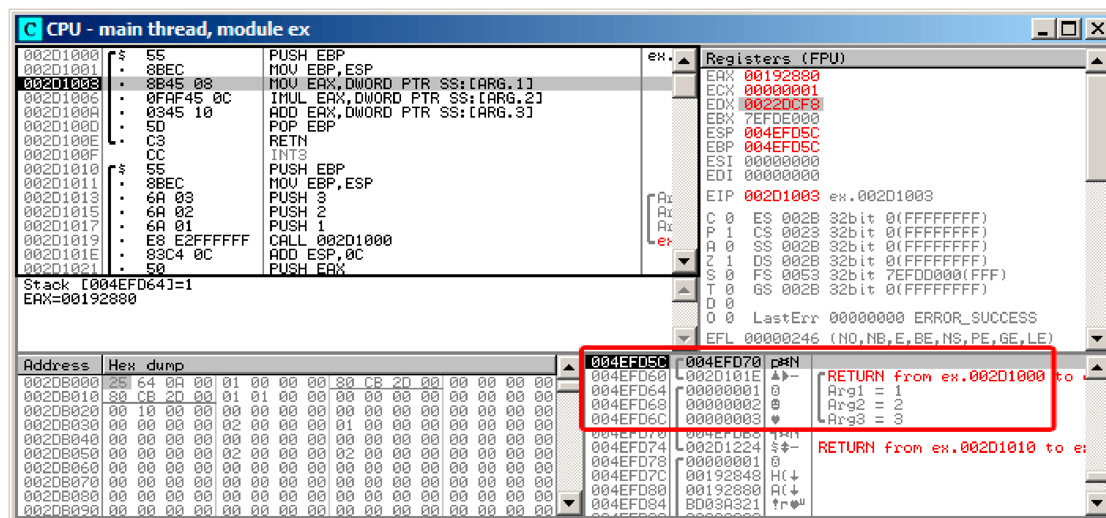
Do odwołania się do pierwszego arugmentu należy dodać dokładnie 8 (dwa 32-bitowe słowa) do EBP.

OllyDbg potrafi to rozpoznać i dodał odpowiednie komentarze do elementów na stosie:

„RETURN from” czy „Arg1 = …”, etc.

Tak naprawdę - argumenty funkcji nie należą do ramki stosu funkcji, są elementami ramki stosu funkcji wywołującej.

Z tego powodu OllyDbg oznaczył argumenty „Arg” jako elementy innej ramki stosu.



Rysunek 1.23: OllyDbg: inside of f() function

GCC

Skompilujmy ten sam przykład w GCC 4.4.1 i podejrzyjmy rezultat w programie IDA:

Listing 1.92: GCC 4.4.1

```
        public f
f       proc near

arg_0   = dword ptr  8
arg_4   = dword ptr  0Ch
arg_8   = dword ptr  10h

        push    ebp
        mov     ebp, esp
        mov     eax, [ebp+arg_0] ; 1 argument
        imul    eax, [ebp+arg_4] ; 2 argument
        add     eax, [ebp+arg_8] ; 3 argument
        pop     ebp
        retn
f       endp
```

```
        public main
main    proc near

var_10  = dword ptr -10h
var_C   = dword ptr -0Ch
var_8   = dword ptr -8

        push    ebp
        mov     ebp, esp
        and     esp, 0FFFFFFF0h
        sub     esp, 10h
        mov     [esp+10h+var_8], 3 ; 3 argument
        mov     [esp+10h+var_C], 2 ; 2 argument
        mov     [esp+10h+var_10], 1 ; 1 argument
        call    f
        mov     edx, offset aD  ; "%d\n"
        mov     [esp+10h+var_C], eax
        mov     [esp+10h+var_10], edx
        call    _printf
        mov     eax, 0
        leave
        retn
main    endp
```

Efekt jest taki sam, z małymi różnicami, które już omówiliśmy wcześniej.

Wskaźnik stosu nie jest przywracany po dwóch wywołaniach funkcji (f oraz `printf()`), ponieważ przedostatnia instrukcja LEAVE (**??** on page ??) zajmie się tym na końcu.

1.14.2 x64

Rzecz wygląda nieco inaczej na x86-64. Argumenty funkcji (pierwsze 4 lub 6) są przekazywane przez rejestry, a więc funkcja wywoływana odczytuje je z rejestrów zamiast ze stosu.

MSVC

Optymalizujący MSVC:

Listing 1.93: Optymalizujący MSVC 2012 x64

```
$SG2997 DB      '%d', 0aH, 00H

main    PROC
        sub     rsp, 40
        mov     edx, 2
        lea     r8d, QWORD PTR [rdx+1] ; R8D=3
        lea     ecx, QWORD PTR [rdx-1] ; ECX=1
        call    f
        lea     rcx, OFFSET FLAT:$SG2997 ; '%d'
        mov     edx, eax
        call    printf
        xor     eax, eax
```

```
        add     rsp, 40
        ret     0
main    ENDP

f       PROC
        ; ECX - 1 argument
        ; EDX - 2 argument
        ; R8D - 3 argument
        imul    ecx, edx
        lea     eax, DWORD PTR [r8+rcx]
        ret     0
f       ENDP
```

Jak widać, funkcja `f()` odczytuje wartości wszystkich argumentów z rejestrów.

Instrukcja LEA została użyta do zrealizowania dodawania, najwyraźniej kompilator uznał, że będzie szybsza niż ADD.

LEA jest również używana w funkcji `main()` do przygotowania pierwszego i trzeciego argumentu funkcji `f()`. Kompilator zdecydował, że będzie to szybsze niż klasyczne załadowanie wartości do rejestru za pośrednictwem instrukcji MOV.

Rzućmy okiem na wynik nieoptymalizującego MSVC:

Listing 1.94: MSVC 2012 x64

```
f               proc near

; shadow space:
arg_0           = dword ptr  8
arg_8           = dword ptr  10h
arg_10          = dword ptr  18h

                ; ECX - 1 argument
                ; EDX - 2 argument
                ; R8D - 3 argument
                mov     [rsp+arg_10], r8d
                mov     [rsp+arg_8], edx
                mov     [rsp+arg_0], ecx
                mov     eax, [rsp+arg_0]
                imul    eax, [rsp+arg_8]
                add     eax, [rsp+arg_10]
                retn
f               endp

main            proc near
                sub     rsp, 28h
                mov     r8d, 3 ; 3 argument
                mov     edx, 2 ; 2 argument
                mov     ecx, 1 ; 1 argument
                call    f
                mov     edx, eax
                lea     rcx, $SG2931    ; "%d\n"
                call    printf
```

```
                ; return 0
                xor     eax, eax
                add     rsp, 28h
                retn
main            endp
```

Wynik kompilacji wygląda dość dziwnie, ponieważ wszystkie 3 argumenty z rejestrów zostały z jakiegoś powodu odłożone na stos.

Nazywamy to „shadow space” [81]:

Każda funkcja w Win64 może (ale nie musi) zapisać tam wartości 4 argumentów, przekazywanych przez rejestry. Dzieje się to z dwóch powodów: 1) przeznaczanie całego rejestru (lub nawet 4 rejestrów) na argumenty jest rozrzutne, więc dostęp do nich będzie zachodził przez stos 2) ułatwia to debuggowanie, gdyż debugger zawsze wie, gdzie znaleźć argumenty funkcji [82].

Czasami duże funkcje mogą zapisywać swoje argumenty do „shadow space”, jeśli będą one wykorzystywane podczas wykonania, ale małe funkcje nie muszą tego robić.

Odpowiedzialnością funkcji wywołującej jest zaalokowanie na stosie miejsca na „shadow space”.

GCC

Optymalizujący GCC generuje dość zrozumiały kod:

Listing 1.95: Optymalizujący GCC 4.4.6 x64

```
f:
        ; EDI - 1 argument
        ; ESI - 2 argument
        ; EDX - 3 argument
        imul    esi, edi
        lea     eax, [rdx+rsi]
        ret

main:
        sub     rsp, 8
        mov     edx, 3
        mov     esi, 2
        mov     edi, 1
        call    f
        mov     edi, OFFSET FLAT:.LC0 ; "%d\n"
        mov     esi, eax
        xor     eax, eax  ; liczba rejestrów wektorowych z argumentami
        call    printf
        xor     eax, eax
        add     rsp, 8
        ret
```

[81] MSDN
[82] MSDN

Nieoptymalizujący GCC:

Listing 1.96: GCC 4.4.6 x64

```
f:
        ; EDI - 1 argument
        ; ESI - 2 argument
        ; EDX - 3 argument
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-4], edi
        mov     DWORD PTR [rbp-8], esi
        mov     DWORD PTR [rbp-12], edx
        mov     eax, DWORD PTR [rbp-4]
        imul    eax, DWORD PTR [rbp-8]
        add     eax, DWORD PTR [rbp-12]
        leave
        ret
main:
        push    rbp
        mov     rbp, rsp
        mov     edx, 3
        mov     esi, 2
        mov     edi, 1
        call    f
        mov     edx, eax
        mov     eax, OFFSET FLAT:.LC0 ; "%d\n"
        mov     esi, edx
        mov     rdi, rax
        mov     eax, 0  ; liczba rejestrów wektorowych z argumentami
        call    printf
        mov     eax, 0
        leave
        ret
```

W System V *NIX ([Michael Matz, Jan Hubicka, Andreas Jaeger, Mark Mitchell, *System V Application Binary Interface. AMD64 Architecture Processor Supplement*, (2013)] [83]) nie ma wymagania o „shadow space”, ale funkcje wywoływane mogą zapisywać swoje argumenty przy niedoborze rejestrów.

GCC: uint64_t zamiast int

Nasz przykład wykorzystuje 32-bitowy typ *int*, dlatego używane są 32-bitowe części rejestrów (z prefiksem E-).

Zmodyfikujmy nieco przykład, by użyć wartości 64-bitowych:

```
#include <stdio.h>
#include <stdint.h>

uint64_t f (uint64_t a, uint64_t b, uint64_t c)
```

[83] Dostęp także przez https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf

```
{
        return a*b+c;
};

int main()
{
        printf ("%lld\n", f(0x1122334455667788,
                            0x1111111122222222,
                            0x3333333344444444));
        return 0;
};
```

Listing 1.97: Optymalizujący GCC 4.4.6 x64

```
f       proc near
        imul    rsi, rdi
        lea     rax, [rdx+rsi]
        retn
f       endp

main    proc near
        sub     rsp, 8
        mov     rdx, 3333333344444444h ; 3 argument
        mov     rsi, 1111111122222222h ; 2 argument
        mov     rdi, 1122334455667788h ; 1 argument
        call    f
        mov     edi, offset format ; "%lld\n"
        mov     rsi, rax
        xor     eax, eax ; liczba rejestrów wektorowych z argumentami
        call    _printf
        xor     eax, eax
        add     rsp, 8
        retn
main    endp
```

Kod jest taki sam, ale tym razem użyto *całych* rejestrów (z prefiksem R-).

1.14.3 ARM

Nieoptymalizujący Keil 6/2013 (tryb ARM)

```
.text:000000A4 00 30 A0 E1      MOV     R3, R0
.text:000000A8 93 21 20 E0      MLA     R0, R3, R1, R2
.text:000000AC 1E FF 2F E1      BX      LR
...
.text:000000B0             main
.text:000000B0 10 40 2D E9      STMFD   SP!, {R4,LR}
.text:000000B4 03 20 A0 E3      MOV     R2, #3
.text:000000B8 02 10 A0 E3      MOV     R1, #2
.text:000000BC 01 00 A0 E3      MOV     R0, #1
.text:000000C0 F7 FF FF EB      BL      f
.text:000000C4 00 40 A0 E1      MOV     R4, R0
.text:000000C8 04 10 A0 E1      MOV     R1, R4
```

```
.text:000000CC 5A 0F 8F E2       ADR     R0, aD_0        ; "%d\n"
.text:000000D0 E3 18 00 EB       BL      __2printf
.text:000000D4 00 00 A0 E3       MOV     R0, #0
.text:000000D8 10 80 BD E8       LDMFD   SP!, {R4,PC}
```

Funkcja `main()` po prostu wywołuje dwie inne funkcje, przekazując trzy wartości do pierwszej z nich —(`f()`).

Jak zauważyliśmy poprzednio, w ARM 4 pierwsze wartości są zwykle przekazywane przez 4 pierwsze rejestry (R0-R3).

Funkcja `f()` używa trzech pierwszych (R0-R2) do przechowywania argumentów.

Instrukcja MLA (*Multiply Accumulate*) mnoży dwa pierwsze operandy (R3 i R1), dodaje do ich iloczynu trzeci (R2) a wynik zapisuje do rejestru zerowego (R0), który, zgodnie ze standardem, służy do zwracania wartości z funkcji.

Jednoczesne mnożenie i dodawanie (*Fused multiply–add*) [84]. jest bardzo użyteczną operacją. Na x86 nie było takich instrukcji przed wprowadzaniem rozszerzenia FMA (zestaw nowych instrukcji typu SIMD) [85].

Pierwsza instrukcja `MOV R3, R0`, jest nadmiarowa (operację można by zrealizować za pomocą tylko jednej instrukcji MLA). Kompilator pominął optymalizację, gdyż pracował w trybie z wyłączoną optymalizacją.

Instrukcja BX zwraca sterowanie do adresu przechowywanego w rejestrze LR i, jeśli trzeba, zmienia tryb pracy procesora z Thumb na ARM bądź odwrotnie. Może się to okazać niezbędne, gdyż funkcja `f()` nie wie z jakiego kodu jest wywoływana, może to być zarówno ARM jak i Thumb. Zatem jeśli jest wywoływana z kodu Thumb, BX zwróci sterowanie do funkcji wywołującej i zmieni tryb procesora na Thumb. Jeśli funkcja została wywołana z kodu ARM, wtedy nie zmieni trybu [*ARM(R) Architecture Reference Manual, ARMv7-A and ARMv7-R edition*, (2012)A2.3.2].

Optymalizujący Keil 6/2013 (tryb ARM)

```
.text:00000098             f
.text:00000098 91 20 20 E0       MLA     R0, R1, R0, R2
.text:0000009C 1E FF 2F E1       BX      LR
```

Widać funkcję `f()` skompilowaną za pomocą kompilatora Keil w trybie z pełną optymalizacją (`-O3`).

Instrukcja MOV została usunięta. Teraz MLA używa wszystkich rejestrów z argumentami i zapisuje wynik do R0, dokładnie tam, skąd funkcja wywołująca tę wartość odczyta.

Optymalizujący Keil 6/2013 (tryb Thumb)

[84] przyp. tłum. - prawdziwe *Fused multiply-add* stosuje jedno zaokrąglanie podczas tej operacji - Wikipedia. Instrukcja MLA nie jest opisana jako *Fused* na stronie Keil

[85] wikipedia

```
.text:0000005E 48 43          MULS    R0, R1
.text:00000060 80 18          ADDS    R0, R0, R2
.text:00000062 70 47          BX      LR
```

Instrukcja MLA nie jest dostępna w trybie Thumb, więc kompilator generuje dwie osobne instrukcje (mnożenie i dodawanie).

Pierwsza instrukcja, MULS, mnoży R0 przez R1 i zapisuje wynik do R0. Druga instrukcja (ADDS) dodaje iloczyn i R2 a wynik zapisuje do R0.

ARM64

Optymalizujący GCC (Linaro) 4.9

Wynik kompilacji jest bardzo prosty. Instrukcja MADD przeprowadza jednoczesne mnożenie i dodawanie (podobnie jak MLA, którą widzieliśmy wcześniej). Wszystkie 3 argumenty są w 32-bitowych częściach rejestrów X, ponieważ są one 32-bitowymi liczbami typu *int*. Wynik jest zwracana przez W0.

Listing 1.98: Optymalizujący GCC (Linaro) 4.9

```
f:
        madd    w0, w0, w1, w2
        ret

main:
; zapisz FP i LR w ramce stosu:
        stp     x29, x30, [sp, -16]!
        mov     w2, 3
        mov     w1, 2
        add     x29, sp, 0
        mov     w0, 1
        bl      f
        mov     w1, w0
        adrp    x0, .LC7
        add     x0, x0, :lo12:.LC7
        bl      printf
; zwróć 0
        mov     w0, 0
; przywróć FP i LR
        ldp     x29, x30, [sp], 16
        ret

.LC7:
        .string "%d\n"
```

Rozszerzmy typ danych do 64-bitowych liczb typu uint64_t:

```
#include <stdio.h>
#include <stdint.h>

uint64_t f (uint64_t a, uint64_t b, uint64_t c)
```

```
{
        return a*b+c;
};

int main()
{
        printf ("%lld\n", f(0x1122334455667788,
                            0x1111111122222222,
                            0x3333333344444444));
        return 0;
};
```

```
f:
        madd    x0, x0, x1, x2
        ret
main:
        mov     x1, 13396
        adrp    x0, .LC8
        stp     x29, x30, [sp, -16]!
        movk    x1, 0x27d0, lsl 16
        add     x0, x0, :lo12:.LC8
        movk    x1, 0x122, lsl 32
        add     x29, sp, 0
        movk    x1, 0x58be, lsl 48
        bl      printf
        mov     w0, 0
        ldp     x29, x30, [sp], 16
        ret

.LC8:
        .string "%lld\n"
```

Funkcja `f()` jest niemal taka sama, używa jednak pełnych 64-bitowych rejestrów X. Duże 64-bit wartości są zapisywane do rejestrów w cześciach, patrz: **??** on page ??.

Nieoptymalizujący GCC (Linaro) 4.9

Kompilator z wyłączoną optymalizacją generuje dłuższy kod:

```
f:
        sub     sp, sp, #16
        str     w0, [sp,12]
        str     w1, [sp,8]
        str     w2, [sp,4]
        ldr     w1, [sp,12]
        ldr     w0, [sp,8]
        mul     w1, w1, w0
        ldr     w0, [sp,4]
        add     w0, w1, w0
        add     sp, sp, 16
        ret
```

Kod zapisuje swoje argumenty wejściowe na stosie lokalnym, na wypadek gdyby coś w funkcji musiało użyć rejestrów `W0...W2`. Dzięki temu unikniemy nadpisywania oryginalnych wartości argumentów funkcji, które mogą być potrzebne w przyszłości.

Jest to nazywane *Register Save Area.* [*Procedure Call Standard for the ARM 64-bit Architecture (AArch64)*, (2013)][86]. Funkcja wywoływana nie ma obowiązku tego robić. Przypomina to „Shadow Space”: 1.14.2 on page 134.

Dlaczego optymalizujący GCC 4.9 pominął kod zapisujący argumenty? Ponieważ przeprowadził analizę i wywnioskował, że argumenty funkcji nie będą potrzebne w przyszłości i rejestry `W0...W2` nie będą używane.

Widać również parę instrukcji `MUL`/`ADD` zamiast pojedynczej `MADD`.

1.15 switch()/case/default

1.15.1

```
#include <stdio.h>

void f (int a)
{
    switch (a)
    {
    case 0: printf ("zero\n"); break;
    case 1: printf ("one\n"); break;
    case 2: printf ("two\n"); break;
    default: printf ("something unknown\n"); break;
    };
};

int main()
{
    f (2); // test
};
```

Wnioski

listing.**??**.

1.15.2 Ćwiczenia

Ćwiczenie#1

Polish text placeholder

[86] Dostęp także przez http://infocenter.arm.com/help/topic/com.arm.doc.ihi0055b/IHI0055B_aapcs64.pdf

1.16 Loops

1.16.1 Ćwiczenia

- http://challenges.re/54
- http://challenges.re/55
- http://challenges.re/56
- http://challenges.re/57

1.17 More about strings

1.17.1 strlen()

```
int my_strlen (const char * str)
{
        const char *eos = str;

        while( *eos++ ) ;

        return( eos - str - 1 );
}

int main()
{
        // test
        return my_strlen("hello!");
};
```

ARM

1.18 Replacing arithmetic instructions to other ones

1.18.1 Ćwiczenie

- http://challenges.re/59

1.19 Arrays

yy [87]

1.19.1

[87] AKA „homogener Container".

```
#include <stdio.h>

int main()
{
    int a[20];
    int i;

    for (i=0; i<20; i++)
        a[i]=i*2;

    for (i=0; i<20; i++)
        printf ("a[%d]=%d\n", i, a[i]);

    return 0;
};
```

1.19.2

1.19.3 Wnioski

Tablica jest zbiorem wartości połozonych obok siebie w pamięci.

Dotyczy to każdego typu elementów, włączając w to struktury.

Aby uzyskać dostęp do konkretnego elementu tablicy, wystarczy obliczyć jego adres.

1.19.4 Ćwiczenia

- http://challenges.re/62
- http://challenges.re/63
- http://challenges.re/64
- http://challenges.re/65
- http://challenges.re/66

1.20 Structures

1.20.1 UNIX: struct tm

1.20.2

1.20.3 Ćwiczenia

- http://challenges.re/71
- http://challenges.re/72

1.21

1.21.1

Rozdział 2

Polish text placeholder

Rozdział 3

Rozdział 4

Java

4.1 Java

4.1.1

4.1.2

4.1.3

Rozdział 5

5.1 Linux

5.2 Windows NT

5.2.1 Windows SEH

SEH

[Matt Pietrek, *A Crash Course on the Depths of Win32™ Structured Exception Handling*, (1997)][1], [Igor Skochinsky, *Compiler Internals: Exceptions and RTTI*, (2012)] [2].

5.3

5.4

Pierre Capillon – Black-box cryptanalysis of home-made encryption algorithms: a practical case study.

How to Hack an Expensive Camera and Not Get Killed by Your Wife.

---

[1] Dostęp także przez `http://www.microsoft.com/msj/0197/Exception/Exception.aspx`

[2] Dostęp także przez `http://yurichev.com/mirrors/RE/Recon-2012-Skochinsky-Compiler-Internals.pdf`

Rozdział 6

Rozdział 7

Książki/blogi warte przeczytania

7.1 Książki i inne materiały

7.1.1 Inżynieria wsteczna

- Eldad Eilam, *Reversing: Secrets of Reverse Engineering*, (2005)
- Bruce Dang, Alexandre Gazet, Elias Bachaalany, Sebastien Josse, *Practical Reverse Engineering: x86, x64, ARM, Windows Kernel, Reversing Tools, and Obfuscation*, (2014)
- Michael Sikorski, Andrew Honig, *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*, (2012)
- Chris Eagle, *IDA Pro Book*, (2011)
- Reginald Wong, *Mastering Reverse Engineering: Re-engineer your ethical hacking skills*, (2018)

(Stara, ale wciąż interesująca) Pavol Cerven, *Crackproof Your Software: Protect Your Software Against Crackers*, (2002).

Oraz książki Krisa Kaspersky'ego.

7.1.2 Windows

- Mark Russinovich, *Microsoft Windows Internals*
- Peter Ferrie – The "Ultimate" Anti-Debugging Reference[1]

Blogi:

- Microsoft: Raymond Chen

[1] http://pferrie.host22.com/papers/antidebug.pdf

- nynaeve.net

7.1.3 C/C++

- Brian W. Kernighan, Dennis M. Ritchie, *The C Programming Language*, 2ed, (1988)
- *ISO/IEC 9899:TC3 (C C99 standard)*, (2007)[2]
- Bjarne Stroustrup, *The C++ Programming Language, 4th Edition*, (2013)
- C++11 standard[3]
- Agner Fog, *Optimizing software in C++* (2015)[4]
- Marshall Cline, *C++ FAQ*[5]
- Dennis Yurichev, *C/C++ programming language notes*[6]
- JPL Institutional Coding Standard for the C Programming Language[7]

7.1.4 x86 / x86-64

- manuale Intela[8]
- manuale AMD [9]
- Agner Fog, *The microarchitecture of Intel, AMD and VIA CPUs*, (2016)[10]
- Agner Fog, *Calling conventions* (2015)[11]
- *Intel® 64 and IA-32 Architectures Optimization Reference Manual*, (2014)
- *Software Optimization Guide for AMD Family 16h Processors*, (2013)

Trochę stare, ale wciąż interesujące:

Michael Abrash, *Graphics Programming Black Book*, 1997[12] (znany z pracy nad niskopoziomową optymalizacją w takich projektach jak Windows NT 3.1 i id Quake).

7.1.5 ARM

- manuale ARM[13]

[2] Dostęp także przez http://www.open-std.org/jtc1/sc22/WG14/www/docs/n1256.pdf
[3] Dostęp także przez http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3690.pdf.
[4] Dostęp także przez http://agner.org/optimize/optimizing_cpp.pdf.
[5] Dostęp także przez http://www.parashift.com/c++-faq-lite/index.html
[6] Dostęp także przez http://yurichev.com/C-book.html
[7] Dostęp także przez https://yurichev.com/mirrors/C/JPL_Coding_Standard_C.pdf
[8] Dostęp także przez http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html
[9] Dostęp także przez http://developer.amd.com/resources/developer-guides-manuals/
[10] Dostęp także przez http://agner.org/optimize/microarchitecture.pdf
[11] Dostęp także przez http://www.agner.org/optimize/calling_conventions.pdf
[12] Dostęp także przez https://github.com/jagregory/abrash-black-book
[13] Dostęp także przez http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.subset.architecture.reference/index.html

- *ARM(R) Architecture Reference Manual, ARMv7-A and ARMv7-R edition*, (2012)
- [*ARM Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile*, (2013)][14]
- Advanced RISC Machines Ltd, *The ARM Cookbook*, (1994)[15]

7.1.6 Język maszynowy

Richard Blum — Professional Assembly Language.

7.1.7 Java

[Tim Lindholm, Frank Yellin, Gilad Bracha, Alex Buckley, *The Java(R) Virtual Machine Specification / Java SE 7 Edition*] [16].

7.1.8 UNIX

Eric S. Raymond, *The Art of UNIX Programming*, (2003)

7.1.9 Programowanie

- Brian W. Kernighan, Rob Pike, *Practice of Programming*, (1999)
- Henry S. Warren, *Hacker's Delight*, (2002). Niektórzy twierdzą, że sztuczki z tej książki nie mają dzisiaj znaczenia, ponieważ miały zastosowanie wyłącznie w procesorach RISC, gdzie instrukcje typu branch są kosztowne. Niemniej jednak, wszystko to znacząco ułatwia zrozumienie algebry Boole'a i całej matematyki wokół tego.

7.1.10

- Bruce Schneier, *Applied Cryptography*, (John Wiley & Sons, 1994)
- (Free) lvh, *Crypto 101*[17]
- (Free) Dan Boneh, Victor Shoup, *A Graduate Course in Applied Cryptography*[18].

7.1.11 Coś jeszcze prostszego

Osobom, dla których ta książka jest zbyt trudna i techniczna, polecam łagodne wprowadzenie do niskopoziomowych zagadnień związanych z maszynami liczącymi: "Code: The Hidden Language of Computer Hardware and Software" Charlesa Petzolda.

[14] Dostęp także przez http://yurichev.com/mirrors/ARMv8-A_Architecture_Reference_Manual_(Issue_A.a).pdf
[15] Dostęp także przez https://yurichev.com/ref/ARM%20Cookbook%20(1994)/
[16] Dostęp także przez https://docs.oracle.com/javase/specs/jvms/se7/jvms7.pdf; http://docs.oracle.com/javase/specs/jvms/se7/html/
[17] Dostęp także przez https://www.crypto101.io/
[18] Dostęp także przez https://crypto.stanford.edu/~dabo/cryptobook/

Inną prostą książką jest komiks dla dzieci[19] z 1983 roku, poświęcony mikroprocesorom 6502 i Z80.

[19] https://yurichev.com/mirrors/machine-code-for-beginners.pdf

Użyte akronimy

OS System operacyjny (Operating System) ix

PL Język programowania (Programming Language) vi

ROM Pamięć tylko do odczytu (Read-Only Memory) 108

ALU Jednostka arytmetyczno-logiczna (Arithmetic Logic Unit) 36

LIFO Ostatni na wejściu, pierwszy na wyjściu (Last In First Out) 42

ABI Interfejs binarny aplikacji (Application Binary Interface) 22

PC Program Counter. IP/EIP/RIP w x86/64. PC w ARM. 26

SP wskaźnik stosu. SP/ESP/RSP w x86/x64. SP w ARM. 26

RA adres powrotu 8

PE Portable Executable (format plików wykonywalnych w systemach Windows) 7

LR Link Register 8

IDA Interaktywny deasembler i debugger rozwijany przez Hex-Rays 7

MSVC Microsoft Visual C++

AKA Also Known As — znany również jako 42

CRT C Runtime library 14

CPU Central Processing Unit ix

CISC Complex Instruction Set Computing 27

RISC Reduced Instruction Set Computing 3

BSS Block Started by Symbol 34

DBMS Database Management Systems vi

ISA Instruction Set Architecture (architektura listy rozkazów) 2

SEH Structured Exception Handling 51

ELF Executable and Linkable Format: Format plików wykonywalnych używany w systemach z rodziny *NIX, w szczególności na Linuksie 106

NOP No Operation 9

BEQ (PowerPC, ARM) Branch if Equal 126

RAM Random-Access Memory 108

GCC GNU Compiler Collection 5

ASCIIZ ASCII Zero ( ) 123

GPR General Purpose Registers (rejestry ogólnego przeznaczania) 2

GDB GNU Debugger 65

FP Frame Pointer 33

STMFD Store Multiple Full Descending ()

LDMFD Load Multiple Full Descending ()

STMED Store Multiple Empty Descending () 42

LDMED Load Multiple Empty Descending () 42

STMFA Store Multiple Full Ascending () 42

LDMFA Load Multiple Full Ascending () 42

STMEA Store Multiple Empty Ascending () 42

Słownik terminów

anti-pattern coś powszechnie uznanego jako zła praktyka. 45, 102

callee funkcja wywoływana. 63, 115, 129, 132, 135

caller funkcja wywołująca. 8–11, 41, 63, 115, 129–131, 134

endianess kolejność bajtów. 30, 105

funkcja liść Funkcja, która nie wywołuje żadnej innej. 39, 45

GiB gibibajt: 2^{10} (1024) mebibajtów, 2^{20} (1048576) kibibajtów lub 2^{30} (1073741824) bajtów. 22

heap (kopiec, sterta) - przeważnie duży kawałek pamięci, zapewniony aplikacji przez OS na jej własne potrzeby. malloc()/free() pracują ze stertą. 43

inkrementować zwiększać o 1. 23

inżynieria wsteczna proces odkrywania jak dana rzecz działa, czasami w celu jej sklonowania. iii

przesunięcie skoku część kodu operacji instrukcji JMP i Jcc, która jest dodawana do adresu kolejnej instrukcji by wyliczyć nową wartość PC. Może mieć wartość ujemną. 124

ramka stosu Część stosu, która zawiera informacje specyficzne dla bieżącej funkcji: zmienne lokalne, argumenty funkcji, RA, etc.. 92, 130

rejestr powrotu (RISC) Rejestr, w który zwykle przechowywany jest adres powrotu. Dzięki temu można wywoływać funkcje-liście (leaf functions) bez używania stosu - a więc szybciej. 44

stdout standardowe wyjście. 29, 49

thunk function prosta funkcja, której jedynym zadaniem jest wywołanie innej funkcji. 31, 58

wskaźnik stosu rejestr pokazujący na miejsce na stosie. 13, 15, 27, 42, 48, 59, 74, 76, 98, 132

Indeks